React.js实战



快速学会React，全面掌握Web前端开发技术

- 针对每个知识点使用示例进行说明，更加注重实践
- 简易笔记本、购物车2个项目案例，快速提升实战能力
- 适合所有想全面掌握React前端开发技术的人员阅读

赵荣娇　刘江虹　著



清华大学出版社

React.js实战



赵荣娇　刘江虹　著

清華大学出版社
北京

内 容 简 介

本书旨在帮读者从零开始学习 React 基础知识，采用“语法”+“示例”的方式，以便于初学者学习和练习，是目前市场上少有的 React 入门图书。

本书共 14 章，分为 3 篇，涵盖的主要内容有：React 的前世今生、使用 React 所需的预备知识（包括 npm、webpack、ES6）、React 开发环境搭建、React 组件、React 事件系统、React 原理、数据管理、React 架构、React 服务端渲染、React 测试、React 性能优化、React+webpack+ES6 项目实战（笔记本+购物车）等。

本书内容丰富、实例典型、实用性强，适合有一定的 HTML、CSS、JavaScript 基础、希望全面学习 React 开发的前端开发人员阅读，也适合希望提高项目开发水平的人员阅读。

图书在版编目（CIP）数据

React.js 实战 / 赵荣娇，刘江虹著.—北京：清华大学出版社，2019

（Web 前端技术丛书）

ISBN 978-7-302-52873-9

Ⅰ. ①R… Ⅱ. ①赵… ②刘… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 083029 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：三河市龙大印装有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**17.25　　**字　　数：**442 千字
版　　次：2019 年 6 月第 1 版　　**印　　次：**2019 年 6 月第 1 次印刷
定　　价：59.00 元

产品编号：082471-01

前　言

随着互联网技术的发展，前端技术的发展也进入一个新的阶段。早期的网页开发是由后端主导的，前端的操作局限于 DOM 区域。随着基础设置的不断完善以及代码封装层级的不断提高，使得前端能够完成的事越来越多，前端所需解决的业务场景也越来越复杂。

近几年，前端已经发展到跨端、跨界面的革新阶段，目前主流以基于 MVVM、Virtual DOM、前后端同构技术进行开发的项目居多，实现的方向也多种多样。React 就是在此基础上发展起来的框架，独特的设计思想所带来的革命性创新让其成为前端新技术的代表。

目前市场上关于 React 开发及实践的图书不少，但真正从零基础搭建开始，通过语法和小示例指导读者提高开发水平的图书却很少。本书便是以实战为主旨，通过 React 开发中所需要涉及的基础知识和两个完整的项目案例，让读者全面、深入、透彻地理解 React 开发的技术栈的整合使用。

本书的技术点

本书涵盖 npm、Node.js、webpack、ES6、React、JSX、Redux、Jest、Enzyme、Hooks、ESLint、Chrome 插件、JavaScript、CSS、ImmutableJS、Perf 等热门技术及整个技术站框架的整合使用。

本书最后使用 React+webpack+ES6 组合形式，开发了笔记本和购物车两个完整项目。读者将案例稍加修改，便可用于实际项目开发实践中。

本书的内容

本书共有 14 章，由浅到深地介绍 React 技术栈中的主要技术，主要内容分为基础篇、进阶篇和实战篇，每一篇内容又分成若干章节来介绍。

第 1 篇　基础篇（第 1～3 章）

介绍 React 的前世今生，以及 React 开发中涉及的基本概念，包括 React 的开发环境和开发工具、React 的基本用法。每个知识点都有配套的源代码示例。

第 2 篇　进阶篇（第 4～12 章）

深入介绍 React 的几个重要概念，包括 React 组件、React 事件系统、React 原理、数据管理、React 架构、React 服务端渲染等。每章都配有大量示例代码，保证读者学以致用。

第 3 篇　实战篇（第 13～14 章）

本篇通过笔记本和购物车两个项目整合使用 React 技术栈，包括 React Router、Redux、SSR，每一个技术都配有详细的项目实战演示。

关于封面照片

封面照片由蜂鸟网的摄影家 ptwkzj 先生友情提供，在此表示衷心感谢。

读者对象

- 有一定的 HTML、CSS、JavaScript 基础的网页开发人员；
- 希望全面学习 React 开发的前端开发人员；
- 希望提高项目开发水平的人员；
- 前端开发培训机构的学员；
- 软件开发项目经理；
- 需要一本案头必备查询手册的人员。

本书作者

本书第 1~6 章由刘江虹完成，第 7~14 章由赵荣娇完成。

著　者

2019 年 5 月

目　　录

第 1 章 React的前世今生

时间追溯到 20 世纪 90 年代，网景浏览器的诞生给互联网世界填写了浓墨的一笔，用户可以在浏览器上查阅信息、分享信息，以及和浏览器进行交互等，当时网景浏览器在市场上的份额是占据第一的。后来微软为了瓜分市场，推出了 IE 浏览器，凭借 Windows 系统的用户量把网景打败。就在微软觉得稳坐天下的时候，Firefox 以及 Chrome 等优秀浏览器的出现，打破了 IE 的市场垄断。可见产品是需要不断创新、不断成长的，故步自封终究会被超越。

JavaScript 的运行环境就是浏览器，其实前端框架也一直处在一个纷争的世界。目前市面上各种 xxx.js 的出现，包括 Ember.js、Angular.js、Backbone.js、Knockout.js、React.js、Vue.js 等，也是纷纷扰扰。前端开发在目前这个互联网环境下还是一个大的趋势，以 JavaScript 为基础语言的前端框架层出不穷，前端技术的更新也非常之快。当今前端框架格局，就上述提到的框架，可以说 Angular、Vue 和 React 三足鼎立。本书主要介绍 React。

1.1 刀耕火种的年代

所谓刀耕火种，就是在 Web 开发早期，技术和工具还不成熟，Web 开发功能和用户体验非常有限，Web 开发人员使用古老的工具耕耘着 Web 这片广阔的天地。

追溯到 Web1.0 时代，当时的 Web 整体架构非常简单，页面基本由 Server 端生成，然后返回给浏览器，页面的呈现也特别粗糙。最初 Web 开发是不分前后端的，所有的逻辑都是在服务器端生成。如果业务逻辑比较简单，那这种方式对于开发人员来说是可以接受的。如果业务逻辑非常复杂，这样会出现很多问题，最大的一个缺点就是关系如果变得复杂，就会导致 Server 非常臃肿，条理不清晰。

直到 MVC 时代的到来，以后端作为出发点，开始细化模块功能，这个时代催生了一些经典的 MVC 框架，比如 Structs、Spring 等。M（Model）层负责数据处理，V（View）层负责界面呈现，C（Controller）层负责处理用户交互功能。这种模式的优点在于开发人员可以各司其职，互不干涉，分工明确。当然也有缺点，View 层和 Controller 层黏度很高，会增加 Web 开发的复杂度。

1.2 Web 应用的出现

有了 Web2.0 这个概念后，互联网开始进入一个新时代。以前用户在浏览器上只能接收文字、图片这样的资讯，处于一个被动接收的角色。在进入 Web2.0 时代后，Web 和用户的交互性加强，类似于桌面程序的 Web 应用大量涌现，这个时期 ajax 的诞生拉开了 SPA（Single Page Application，单页面应用）序幕。

另外，多媒体的诞生也催生了 Web2.0 时代的发展，像音频、视频、Flash 的出现，可以让网页变得更加绚丽多彩，给用户在视觉和听觉上都带来了不一样的体验。可以说 Web2.0 时代的到来给 Web 前端这一块带来了空前的繁荣。

1.3 React 的诞生

React 出生在 Facebook。当初 Facebook 要搭建一个 Instagram 网站，在选择框架时，对市场上的所有 JavaScript 的 MVC 框架都不太满意，于是 Facebook 自己搞了一套，就是 React。后来发现 React 框架非常好用，便在 2013 年 5 月开源了。

React 的出现给 Web 前端开发人员带来了福音，其新颖的创新思路，加上极佳的性能，广受前端开发人员的欢迎。下一章将从零开始讲解 React 的环境搭建、使用方法以及如何设计的相关知识。

在学习 React 之前，读者应该对以下知识进行了解：

- Node.js：基于 Chrome V8 引擎的 JavaScript 运行环境，使用了一个事件驱动、非阻塞 I/O 的模式，使其轻量又高效。
- npm：node 的一个包管理工具，主要功能是对 node 包的安装、卸载、更新、查看等。
- webpack：前端资源加载/打包工具，可以依据模块的依赖关系进行静态分析，按照一定规则将这些模块生成静态资源。
- ES6：也称为 ES2015，是在 2015 年 6 月发布的 JavaScript 语言的下一代标准，旨在编写大型应用程序，成为一种企业级开发语言。



如果读者对 ES5 比较熟悉，可以使用 ES5 进行 React 开发，但是 React 推荐使用 ES6，这在以后是一个趋势，目前 Facebook 官方推荐的标准是 ES6。

1.4 npm

在使用任何一个框架之前，必然要经历的一个环节是环境搭建，而 npm 是配置 React 环境的必要工具，其在下载各种依赖包时起着重要作用。本节主要为读者解析 npm 是怎样的一个工具。

1.4.1　什么是 npm

简单来说 npm（Node Package Manager）是包含在 Node.js 里面的一个包管理工具，如果读者之前使用过 Node.js，那对 npm 应该不会陌生，因为 npm 会随着 Node.js 一起安装。npm 是世界上最大的软件注册表，其为开发者连接到了一个广阔的 JavaScript 世界。据官方数据统计，npm 大约每周有 30 亿的惊人下载量，其中包含大约 60 万个 package（代码模块）。

npm 为开发者提供了一个代码模块共享的大平台，开发者既可以从 npm 服务器上下载其他开发人员共享的 package，也可以上传自己的 package 供其他开发者使用。Node.js 和 npm 的环境搭建将在 2.2 节中详细讲述。

1.4.2　理解 npm scripts

npm scripts 指的是 npm 脚本，其主要用途是执行配置文件（package.json）中“scripts”属性对应的脚本语句。在理解 npm scripts 之前，这里先介绍一下 package.json 文件。

在搭建一个前端项目时，一般在项目的根目录下要生成一个 package.json 文件，该文件用来定义项目信息、配置包依赖关系。package.json 文件可以自己手动创建，也可以用如下命令创建：

```
$ npm init
```

这里列举一个简单的 package.json 文件，如下所示：

```
{
  "name": "ReactDemo",
  "version": "0.1"
}
```

上述 package.json 文件只定义了项目名称和项目版本号。一般情况下，在实际开发中，package.json 文件是非常丰富的，接下来列举一个实际开发中比较全面的 package.json 文件，如下所示：

```
{
  "name": "demo01",
  "version": "0.1.0",
```

```
    "private": true,
    "dependencies": {
      "React": "^16.2.0",
      "React-dom": "^16.2.0",
      "React-scripts": "1.1.1"
    },
    "scripts": {
      "start": "React-scripts start",
      "build": "React-scripts build",
      "test": "React-scripts test --env=jsdom",
      "eject": "React-scripts eject"
    },
    "devDependencies": {
      "prop-types": "^15.6.1"
    }
  }
```

上述 package.json 文件内容多了几个字段，private、dependencies、devDependencies 和 scripts。private 指包的私有属性，如果设置为 true，则 npm 会拒绝发布，主要是为了避免私有 repositories 不小心被发布出去。dependencies、devDependencies 两个字段在 1.4.3 小节介绍，这里主要介绍 scripts。

scripts 里面放的是 npm 要执行的命令，格式是 key-value 形式，为了简化操作，具体命令为 value，自定义的简化命令为 key，当 npm 运行 key 命令时，等同于执行后面的 value 命令。例如，执行 npm run start 命令，相当于执行了 React-scripts start。

简而言之，package.json 配置文件中的脚本，就叫作 npm scripts。其实 scripts 里面的命令可以是任何的 shell 命令，执行 npm run 的时候，会自动构建一个 shell，脚本都是在 shell 中执行，所有 package.json 中的脚本可以是任何可以在 shell 中有效执行的命令。

也许有读者会有疑问，为什么 scripts 命令可以直接使用。这里解释一下，npm run 在新建一个 shell 的时候，会将当前目录的 node_modules/.bin 目录配置到 path 环境变量中，如果以前用过 Java，应该了解在配置 jdk 时需要将 jdk 目录配置到 path 环境变量中才可以全局使用。这里 npm 是自动将 node_modules/.bin 配置到了环境变量中。其实上面的 scripts 字段可以改写为下面的样子：

```
  "scripts": {
      "start": "./node_modules/.bin/React-scripts start",
      "build": "./node_modules/.bin/React-scripts build",
      "test": "./node_modules/.bin/React-scripts test --env=jsdom",
      "eject": "./node_modules/.bin/React-scripts eject"
    },
```

1.4.3　dependencies 和 devDependencies

在 1.4.2 小节中介绍 package.json 文件的内容时提到了 dependencies 和 devDependencies 两个字段：

```
"dependencies": {
    "React": "^16.2.0",
    "React-dom": "^16.2.0",
    "React-scripts": "1.1.1"
  },
//…
"devDependencies": {
    "prop-types": "^15.6.1"
  }
```

dependencies 和 devDependencies 两个配置字段都是用来安装依赖包的，区别在于前者安装项目运行所依赖的模块，后者安装项目开发所依赖的模块。

在 npm 安装模块的时候，会有两个命令：

```
$ npm install <package-name> --save
```

和

```
$ npm install <package-name> --save-dev
```

第一个命令是用来对应 dependencies 字段的，第二个命令是对应 devDependencies 字段的。上述示例中有些模块的版本号之前有个插入号“^”，比如"React": "^16.2.0"，表示安装 React 的 16.x.x 的最新版本（不低于 16.2.0），但是不安装 17.x.x，也就是说安装时不改变大版本号。如果版本号前面没有任何标识，比如"React-scripts": "1.1.1"，表示只安装 React-scripts 的 1.1.1 版本。



有时版本号前面会有波浪“~”，例如“~2.2.3”，表示安装 2.2.x 的最新版本号（不低于 2.2.3），但是不安装 2.3.x，也就是说安装时不改变大版本号和次版本号。另外，需要注意的是，如果大版本号为 0，“~”和“^”的表示作用是一样的。

1.5　webpack

在没有出现模块管理器之前的前端开发，如果要引用依赖资源，通常的做法是将依赖文件引用到.html 文件中。比如，要引用 js 文件，在.html 文件中用<script>标签引用；引用.css 文件，在.html 文件中用<link>标签引用。这样做的弊端是，如果引用的资源文件太多，请求太多，会拖慢网页加载速度，影响用户体验，另外也会使网页体积臃肿、不便维护。随着模块管理器

的出现，上述问题得到解决。目前市面上流行的包管理器有很多，比如 Bower、Browserify、webpack 等，本书主要讲解 webpack 这个前端的包管理器，其他工具的用法和 webpack 大同小异，读者自行网上学习即可。



之前引用的.js 文件或.css 文件即可理解为模块。

webpack 主要有 4 个核心概念：

- 入口（entry）：webpack 所有依赖关系图的起点。
- 出口（output）：指定 webpack 打包应用程序的地方。
- 加载器（loader）：指定加载的需要处理的各类文件。
- 插件（plugins）：定义项目要用到的插件。

1.5.1 为什么需要 webpack

移动互联网时代的网站正在慢慢演化为 Web APP，这种局势愈演愈烈，读者应该能感受到 Web 前端发展之迅速，浏览器在不断强大，JavaScript 的新标准 ES6 也在 2015 年制定，各种流行的 JS 框架问世，发展着实快。另外，现在的 Web 前端更倾向单页面应用（SPA），要求的页面刷新越来越少，这样庞大的代码量如果管理不好就会导致很多的后续问题，比如各个模块耦合度变高、很难维护等。这样就催生了模块管理器。

webpack 是前端的一个模块管理工具，其可依据各个模块之间的依赖关系进行静态分析，然后将这些模块按照相应规则生产静态资源供项目调用。可以通过图 1-1 来理解 webpack 是做什么的。

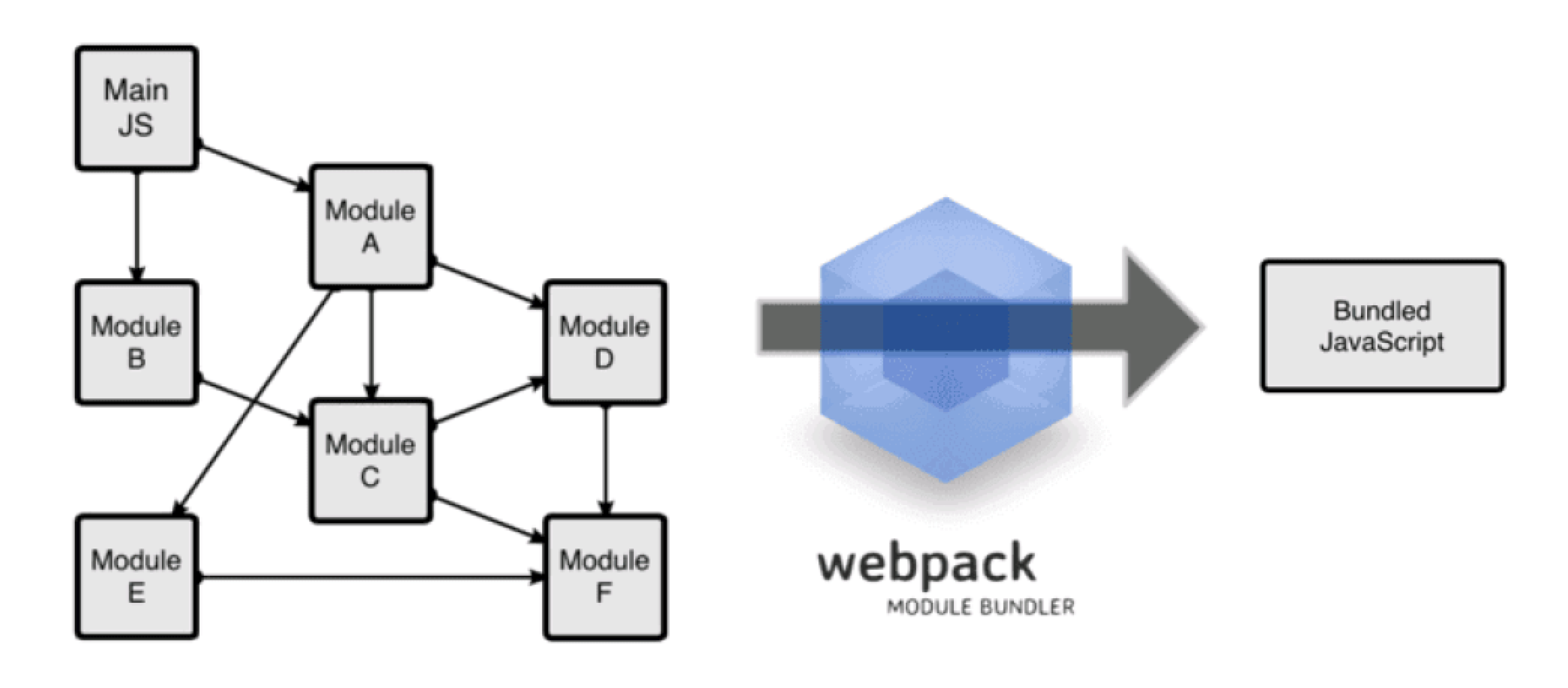


图 1-1 webpack 示意图

可以看出，webpack 可以将具有各种依赖关系的静态模块转化成一个独立的静态模块，这样做的好处是大大减少了请求次数，提高了网页的性能，用户体验也随之提高。

webpack 的另一个作用是可以把目前一些浏览器解释不了的语言进行编译，转换成浏览器可以识别的内容。React 的所有代码示例都以 ES6 标准讲解，ES6 的有些语法目前在一些主流

的浏览器上还不支持，需要对 webpack 进行一些配置后，React 才可正常运行。

1.5.2　webpack 入口和出口

webpack 的四大要素即 entry、output、loader 和 plugins，在讲解这几个要素之前，先了解怎么安装 webpack。

webpack 的安装需要 npm 来完成，有两种安装方式，命令如下：

```
$ npm install -g webpack             //全局安装
```

或者

```
$ npm install --save-dev webpack          //安装到项目目录中
```

安装好之后，就可以使用 webpack 的命令了。比如现在有一个 main.js 文件，要将其打包生成一个 bundle.js 文件，就可以用下面的命令：

```
$ webpack main.js bundle.js
```

一般在实际的项目开发中，要把这些命令写到一个名为 webpack.config.js 的文件中。

webpack 需要处理具有依赖的各个模块，这些模块会构成一个关系图。webpack 的入口就是这张关系图的起点，指的就是入口文件。webpack 的出口指的是需要把这张关系图导出到哪个文件中，即导出文件。这里以一个具体的 webpack.config.js 文件讲解，配置如下：

```
module.exports = {
  entry: './main.js',
  output: {
    filename: 'bundle.js'
  }
};
```

上述 webpack.config.js 文件只配置了项目的 entry 和 output。在该项目的关系图中，main.js 是起点，main.js 可能和别的模块存在依赖关系，但是开发者不需要关心这些，寻找依赖、解决依赖是 webpack 的工作。

entry 字段指定入口文件，也可以理解为 APP 启动时运行的第一个文件。其语法如下：

```
entry: string | Arrary<string>
```

entry 字段可以为一个字符串，也可以是一个字符串数组，所以 entry 可以指定一个入口文件，也可以指定多个入口文件。

output 主要是告诉 webpack 把整理后的所有资源都放在哪里，指定输出位置。上述示例中，output 只指定了 filename，其实还可以指定路径 path，如果省略 path 参数，将默认输出到 webpack.config.js 同级目录下。

1.5.3 webpack loader

webpack 要完成的任务是把具有依赖关系的各个文件进行整合，然后打包，当然文件类型会有很多种，比如 .css、.html、.scss、.jpg 等。但是 webpack 只认识 JavaScript，那处理其他类型的文件是如何做到的呢？loader 解决了这个问题。

loader 在构建文件过程中起着重要的作用，首先 loader 可以识别出要对哪些文件进行预处理，然后 loader 转换这些文件添加到 bundle（构建后的模块）中。例如，React 开发一般使用 JSX 这种扩展语言来编写，JSX 这种格式目前的浏览器是理解不了的，那 webpack loader 可以在 JSX 被项目使用之前做一些预处理，可以将其转换为 JavaScript 语言。示例如下：

```
module.exports = {
  entry: {
    app: './app.js'
  },
  output: {
    filename: 'bundle.js',
    path: './dist'    //输出路径
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        use: 'babel-loader'
      },
      {
        test: /\.css$/,
        use:‘css-loader’
      }
    ]
  }
}
```

上述示例中，test 字段表示要对哪些文件进行构建，use 字段表示要用哪些模块对类型文件进行构建。在配置 loader 之前，use 中的模块是需要安装的。命令如下：

```
$ npm install --save-dev babel-loader
$ npm install --save-dev css-loader
```



在 webpack 早期版本中，写法是这样的：

```
module.exports = {
  //…
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader:'css-loader'
      }
    ]
  }
}
```

webpack 最新版本已废弃了 loaders、loader 的写法，改成了 rules、use。这个读者要注意，网上的一些教程是以老版本的标准写的，如果用老版本的写法，那么在运行 webpack 的时候会报错。

1.5.4　webpack plugins

插件的意义一般是用于丰富功能，webpack 的 plugins 就是用来丰富 webpack 功能的。plugins 在 webpack 中起着重要作用，开发者可以在 webpack.config.js 配置文件中添加想要的其他插件功能。

webpack plugins 的用法很简单，先调用 require，然后在 plugins 字段中用 new 来定义。



插件分为 webpack 自带和第三方这两类，如果是第三方插件，需要在 require 之前利用 npm 安装。

例如，现在需要安装一个第三方插件 html-webpack-plugin，操作步骤如下。

（1）该插件为第三方插件，首先需要 npm 安装，命令为：

```
$ npm install --save-dev html-webpack-plugin
```

（2）然后配置 webpack.config.js 文件中的 plugins：

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
module.exports = {
  entry: {
```

```
      app: './app.js'
    },
    output: {
      filename: 'bundle.js',
      path: './dist'    //输出路径
    },
    module: {
      rules: [
        {
          test: /\.(js|jsx)$/,
          use: 'babel-loader'
        },
        {
          test: /\.css$/,
          use:'css-loader'
        }
      ]
    },
  plugins: [new HtmlWebpackPlugin()]
  }
```

plugins 参数为一个数组，可以传入多个 plugin，另外需要注意 plugin 是可以带参数的，所以 plugins 属性传入的必须为 new 实例。

1.6 ES6

ECMAScript6（简称 ES6）历时将近 7 年时间，在 2015 年 6 月份正式发布，由于这个新标准是在 2015 年指定，所以 ES6 也称为 ES2015。ES6 带来很多新语法、新特性，比如箭头函数（=>）、class（类）、模板字符串等。

相对 ES5（2009 年指定的 ECMAScript 标准）来说，ES6 旨在以新语法和新特性来提高 ECMAScript 的开发效率，ES6 的别名被定义为 Harmony（和谐），读者应该也能体会到，ES6 注定要带来一次优雅的编程变革，本节主要介绍 ES6 的一些新特性。

1.6.1 函数的扩展

ES6 函数扩展的新特性给编程人员和阅读代码人员带来了很多便利之处，以前 ECMAScript 的函数形式看起来有点丑陋，并且在一些使用方面限制也较多。本小节主要讲述 ES6 在函数上都做了哪些扩展。

1. 函数参数默认值

在前端开发中，有时候需要写一些组件供其他开发者调用，这时需要提供对外接口。对外接口参数的传入由调用者决定，如果在调用对外接口时，没有传参，该怎么处理？ES6 之前的函数参数默认值是这样实现的，代码如下：

【示例 1-1】

```
function showName(arg){
    var name = arg || "React";
    console.log(name);
}
showName()                          //输出 React
showName("liujianghong")            //输出 liujianghong
```

以前只能以这种变通的方式来帮助函数处理参数默认值，ES6 提供了新的语法标准，使得函数参数默认值的处理变得简洁，代码如下：

【示例 1-2】

```
function showName(arg="React"){
    console.log(arg);
}
showName()                      //输出 React
showName("liujianghong")        //输出 liujianghong
```

2. 剩余（rest）参数

ES6 发布之前，ECMAScript 对函数的定义中存在一个 arguments 对象，该对象可以访问传入的参数列表。例如，要实现一个求和函数，以前 ECMAScript 的写法是这样的：

【示例 1-3】

```
function sum() {
    var sum = 0;
    for (var val of Array.prototype.slice.call(arguments)) {
        sum += val;
    }
    return sum;
}
console.log(sum(1,2,3))                 //输出 6
```

上面示例是 ES6 之前实现求和功能的写法。



arguments 是一个类数组对象，不是一个真正的数组，所以需要使用 Array.prototype.slice.call()方法将其转换成数组对象。

ES6 在剩余参数上提供了新的语法标准，就上述求和功能，ES6 可以这么写：

【示例 1-4】

```
function sum(...values) {
  let sum = 0;
  for (var val of values) {
    sum += val;
  }
  return sum;
}
console.log(sum(1, 2, 3))            // 输出 6
```

剩余参数的语法为：

```
Function fnName([arg, ]…restArgs){}
```

上述示例中，...values 为剩余参数，调用该函数时，参数个数可以是不确定的。剩余参数是一个真正的数组，所以不需要转换，而且所有的数组特性剩余参数都具有。



在使用剩余参数语法时，剩余参数后面是不可以再有参数的，否则会报错，比如“Function fn1 (…rest, arg){} //”这种写法是错误的。

3. 箭头函数

所谓箭头函数，就是利用箭头（=>）来定义函数，属于匿名函数一类，如果读者了解过 CoffeeScript（JavaScript 的衍生语言），就不会对箭头函数陌生了。由于箭头函数在实现一些功能上比较简洁方便，所以这一个特性的使用率非常高。

箭头函数的语法也非常简单：

```
语法：arg => statement
```

箭头前面是参数，后面是实现语句。例如：

```
var name = function(arg){
 return arg;
}
```

上述功能用 ES6 的箭头函数可以写为：

```
var name = arg => arg;
```

如果参数有多个，并且实现语句也有很多，那参数需要用小括号“()”括起来、用逗号隔开，多条实现语句用大括号“{}”括起来。假如要实现一个多数求和功能，可以这么写：

```
var sum = (num01,num02) => {return num01+num02};
```



箭头函数中大括号中的内容是被解释为代码块的，如果在大括号中返回一个对象，需要在对象外面再加一个小括号，例如：

```
let person = (age,name) => { age: age, name: name };          // 错误
let person = (age,name) => ({ age: age, name: name });        // 正确
```

1.6.2　对象的扩展

ES6 之前的 ECMAScript 虽然说是面向对象语言，但其并没有像 Java 语言那样有类的概念，所以之前的 ECMAScript 编程不是完全面向对象编程，在对象的各种功能方面比较弱化。ES6 标准给予了对象一些扩展，极大地提高了 ECMAScript 对象的可操作性。

1.属性简化

项目中会经常遇到这种情况，一个函数的返回值有多个，以前的前端工程师可能会利用对象字面量或数组来模拟该功能，代码如下：

【示例 1-5】

```
function f(age, name) {
  return {myAge: age, myName: name};
}
console.log(f(26,"liujianghong"));
//输出结果为：{myAge: 26, myName: "liujianghong"}
```

ES6 在表达式的结构上提供了简化功能，上面的例子用 ES6 的标准可以写为：

【示例 1-6】

```
function f(age, name) {
  return {age,name};
}
console.log(f(26,"liujianghong"));
//输出结果为：{age: 26, name: "liujianghong"}
```

从上述示例可以看出，ES6 是允许在对象中直接写变量的，属性名即为变量，属性值即为变量值，这种表达式的简化功能可以使项目代码变得简洁漂亮。

其实在 ES6 标准中，除了字面量属性可以简写外，方法也可以简写。代码如下：

【示例 1-7】

```
//之前写法
const person = {
  showName: function() {
    return "liujianghong";
  }
```

```
};
console.log(person.showName())       //输出结果为：liujianghong
```

与 ES6 的写法作为对比，代码如下：

【示例 1-8】

```
const person = {
  showName() {
    return "liujianghong";
  }
};
console.log(person.showName())       //输出结果为：liujianghong
```

2. 属性名表达式

属性名用表达式代替，这个功能其实在 ES6 之前是有支持的，例如：

```
obj['a'+'bc']='React'
```

如果对象是用字面量来定义，那么这种属性名的表达式是不允许的。ES6 扩展后，这个语法被引入到了对象的字面量中，例如：

【示例 1-9】

```
let propKey = 'foo';
let obj = {
   [propKey]: true,
   ['a' + 'ge']: 26
};
console.log(obj);            //输出结果为：{foo: true, age: 26}
```

在 ES6 中，这种方式的定义也适用于函数名的定义，例如：

【示例 1-10】

```
let obj = {
  ['show' + 'age']() {
    return 26;
  }
};
console.log(obj.showage())   //输出结果为：26
```



属性名表达式和属性简化是不可以同时使用的，例如：

```
// 错误
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] };
// 正确
const foo = 'bar';
const baz = { [foo]: 'abc'};
```

1.6.3　class

有一定 JavaScript 经验的读者应该知道，JavaScript 是可以面向对象编程的一门语言。但是 ES6 之前的 JavaScript 没有原生的类机制，不像 Java、C++语言那样，直接可以用关键字 class 来定义一个类，并且类可以继承、重载等。那以前 JavaScript 没有 class 关键字，类概念是怎么实现的呢？

以前工程师在实现类概念时常常用函数原型来实现类系统，比如要定义一个 Person 类，可以这么写：

【示例 1-11】

```
function Person(name,age,sex){
   this.name = name;
   this.age = age;
   this.sex = sex;
}
Person.prototype.showName = function () {
   console.log(this.name);
}
var p = new Person("xiaoming",26,"man");
p.showName();                //输出结果为：xiaoming
```

ES6 标准中类的基本语法为：

```
class name {…}
```

上述的示例用 ES6 的标准可以改写成这样：

【示例 1-12】

```
class Person{
  constructor(name,age,sex){
      this.name = name;
      this.age = age;
      this.sex = sex;
  }
```

```
    showName(){
        console.log(this.name);
    }
}
const  p = new Person("xiaoming",26,"man");
p.showName();            //输出结果为: xiaoming
```

第 2 章

初探React

通过 1.3 节的讲述，读者应该能够知道，React 是 Facebook 内部构建的一个框架，旨在利用组件化思想让前端 View 层有更好的建设性。本章将带领读者真正进入 React 世界，探寻这个被当代前端开发者追捧的框架的魅力所在。

2.1 React 带来的变化

React 颠覆了以前我们了解的一些 JS 框架（比如 jQuery）的写法和架构，本节就来说说 React 的特色。

2.1.1 React 的声明式编程

声明式编程是告诉机器想要什么信息，机器就返回什么信息，偏重结果。声明式编程可以和命令式编程做一个对比，命令式编程是命令机器要做什么事，机器就做什么事，偏重于过程。为了更好理解这一概念，下面列举两个示例进行对比。

【示例 2-1　命令式编程】

```
<script>
    var arry = [1,2,3,4,5]
    var doubled = []
    for(var i = 0; i < arry.length; i++) {
        var newArry = arry[i] * 2
        doubled.push(newArry)
    }
    console.log(doubled)                    //输出结果为：doubled [2,4,6,8,10]
</script>
```

上述示例中实现的功能为，将 arry 数组中的所有值进行乘 2 操作，运用 for 循环语句来告诉机器，要用这种方式来实现。这就是命令式编程的方式。接下来看声明式编程是怎么实现的。

【示例 2-2　声明式编程】

```
<script>
```

```
    var arry = [1,2,3,4,5]
    var doubled = arry.map(function (val) {
        return val*2;
    })
    console.log(doubled) //=> [2,4,6,8,10]
</script>
```

上述示例的实现，是直接将 arry 数组的所有值进行乘 2 操作后直接将结果返回给 doubled 数组，即告诉机器，要得到一个乘 2 后的结果数组。这就是声明式编程。

React 的声明式编程就是这个原理，把相关的实现都抽离出来，使开发者更多地去关注想要什么。在声明式编程的思想中，更加突出的是整体性的编程思路，这也是 React 这个框架中的一个核心思想。

2.1.2 React 的组件化思想

以前读者在学习 HTML 标签的时候，其实就已经接触到组件化了，HTML 的标签就可以理解为一个组件，比如一个<button></button>，就可以理解成这是一个按钮组件。React 的整体设计思路就是实现自定义的组件。组件化的编程有很多优势：一个好的组件可以在项目中多处使用，这样会节省很多重复工作；其次，组件的分离可以让开发者更加专注每个组件内部的实现，这种高内聚的特性还不会影响到其他开发者的代码模块。

【示例 2-3　React 自定义组件】

```
<script type="text/babel">
    var MyButton = React.createClass({
        render: function () {
            return (<button>This is my button</button>);
        }
    });
    ReactDOM.render(
        <MyButton />,
        document.getElementById('example')
    );
</script>
```

在上述示例中，MyButton 就是一个自定义组件。组件的定义也可以用 ES6 提供的 class 来定义，上述的示例用 class 来定义也可以写为这样：

```
class MyButton extends Component {
    render() {
        return (
            <div className="App">
                <button>This is my button</button>
            </div>
```

```
        );
    }
}
ReactDOM.render(<MyButton />, document.getElementById('example'));
```

2.1.3　React 的虚拟 DOM

在没有虚拟 DOM 概念之前，如果要对 DOM 节点执行修改操作，就会直接操作真实 DOM 树。众所周知，前端中一些性能低的问题，有很多原因就是真实 DOM 树操作太过频繁而导致页面性能下降造成的。

React 中虚拟 DOM 概念的出现，在提高页面性能方面起到了很大的作用。比如现在要操作一个 A 节点，进行三次状态改变：第一步改为 1 状态，第二步改为 2 状态，第三步再改回 1 状态。如果是传统的 DOM 操作，会进行 3 次改变，而利用 React 的虚拟 DOM 技术，会对 A 节点的第一次状态和最后一次状态进行对比。在该例中，第一次和最后一次的状态一样，所以 React 是不会对真实的 DOM 树进行改变的，这样就大大节省了渲染成本，提高了页面运行效率。

也就是说，虚拟 DOM 树可以记录节点的变化过程，但最后真实的渲染结果是由 diff 算法（第 5 章将讲解）来控制的，React 只对有真正变化的节点进行渲染。

2.2　本地环境搭建

在使用 React 之前，需要提前搭建 React 的开发环境。React 有 3 种搭建环境的方式。

（1）引用 React CDN 资源：这种方式通过<Script></Script>标签引用后即可使用。

```
<script crossorigin
src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```

（2）通过 npm 安装 React：这种方式使用包管理工具 npm 安装 React。

```
$npm install --save react react
$ npm install --save react react-dom
```

（3）利用脚手架 create-react-app 来安装 React。

```
$ npm install -g create-react-app
$ create-react-app my-app
$ cd my-app/
$ npm start
```

第 3 种环境搭建方式适合 React 开发初学者，故本书以该方式讲解 React 的环境搭建。

Facebook 为了简化开发者的环境搭建过程，提供了一个脚手架 create-react-app，其提供了项目开发中通用的包依赖，开发者在利用 create-react-app 创建项目时，该脚手架会自动下载所有的依赖包，轻松地搭建 React 开发环境。目前 create-react-app 是市面上最流行的一个脚手架。

2.2.1　Node 与 npm 安装

在讲述 Node 及 npm 安装之前，先带领读者熟悉一下这两种技术。npm 已在 1.4 节中讲过，读者可以返回查阅。下面简单介绍一下 Node。

2009 年 Ryan 正式推出了基于 JavaScript 语言和 V8 引擎的开源 Web 服务器项目，命名为 Node.js，简称 Node。Node 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，使用事件驱动、非阻塞的 I/O 模型，使其轻量而高效。简单来说，Node 是运行在服务器端的 JavaScript。

利用 create-react-app 搭建 React 开发环境之前，需要安装 Node 和 npm，以前这两个包需要分别安装，后来 npm 注入 Node 中，所以现在安装了 Node，就默认安装了 npm。

Node 官网下载链接为 https://nodejs.org/en/download/current/。通过此链接进入官网，可以看到各个系统的 Node 版本，如图 2-1 所示。

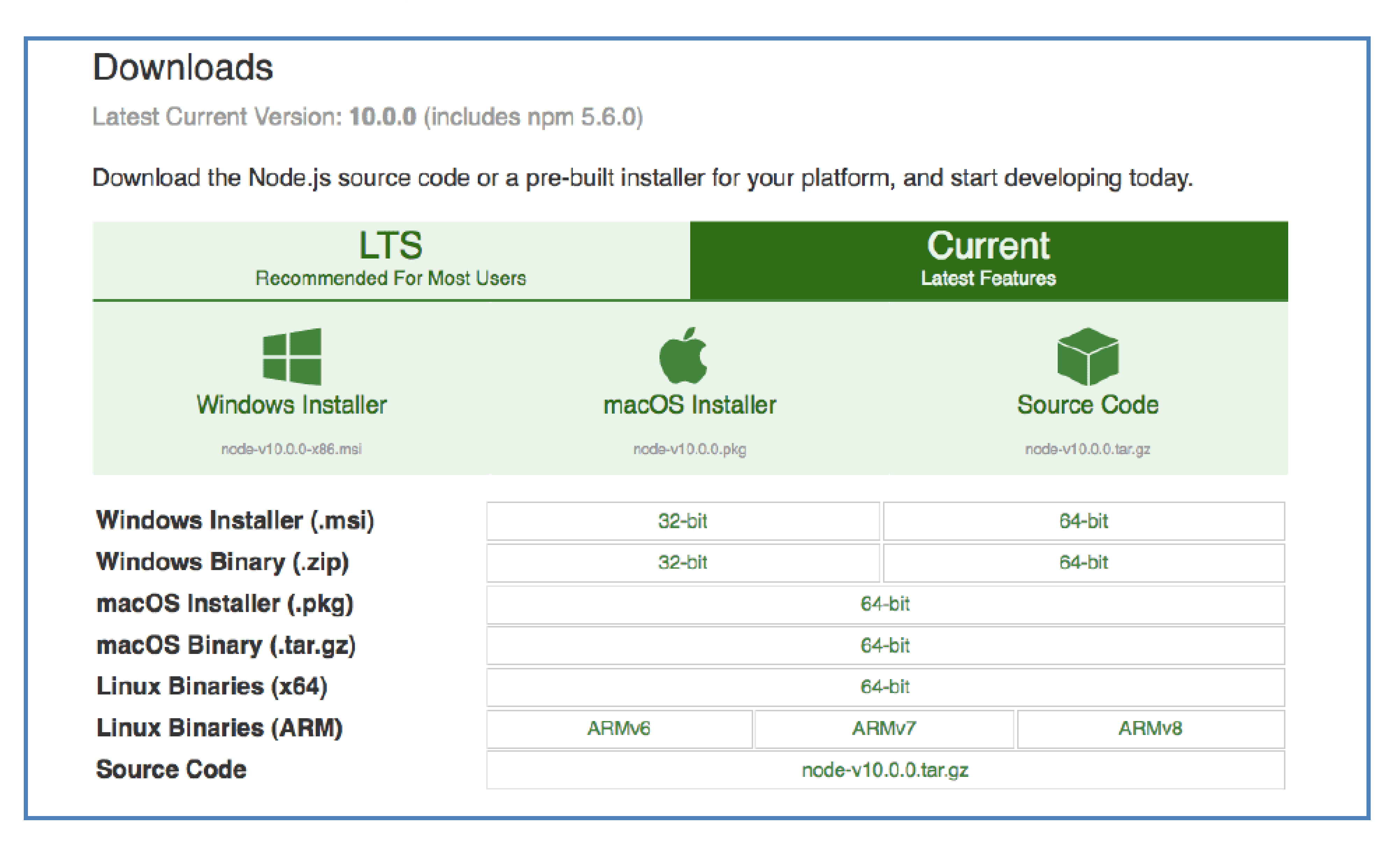


图 2-1　Node 下载页面

目前 Node 最新并且稳定的版本为 10.0.0。读者可以根据自己的系统情况对应下载 Node 并安装。安装完毕后，可通过查看 Node 版本号来确认是否安装成功，在终端输入命令“node –v”即可查看，如图 2-2 所示。

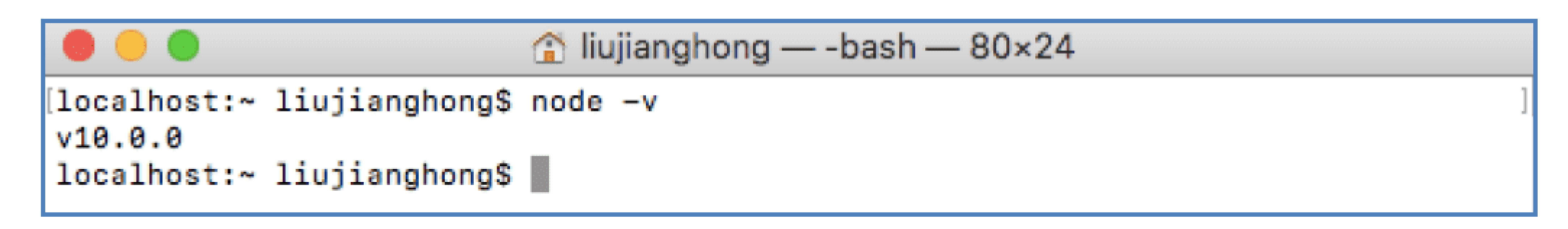


图 2-2　查看 Node 版本号



如果读者电脑曾安装过版本比较低的 Node，想把 Node 升级为最新最稳定的版本，这里给读者一个小小的提示。不需要卸载后再安装，只需要执行以下命令即可：

```
$ npm cache clean -f     //清除 Node 的 cache
$ npm install -g n       //安装 n 工具，该工具是专门管理 Node 版本的工具
$ n stable               //安装最新最稳定的 Node 版本
```

由于 Node 自带 npm，所以安装 Node 后，npm 也已安装。查看 npm 是否安装成功，也可以通过查看其版本号来验证，命令为“npm –v”，如图 2-3 所示。

```
liujianghong — -bash — 80×24
localhost:~ liujianghong$ node -v
v10.0.0
localhost:~ liujianghong$ npm -v
5.6.0
localhost:~ liujianghong$
```

图 2-3　查看 npm 版本号

2.2.2　打造属于你的编辑器

俗话说，工欲善其事，必先利其器。一个优秀的开发工具能够极大地提高前端工程师的开发效率。开发前端的 IDE 有很多，比如 Dreamweaver、Sublime、IntelliJ IDEA、WebStorm 等，就目前各大互联网公司的前端开发环境来说，多数前端工程师选用的是 WebStorm，所以本书主要介绍 WebStorm 开发工具。

WebStorm 是 JetBrains 公司旗下一款 JavaScript 开发工具，被国内前端工程师称为“前端开发神器”。与 IntelliJ IDEA 同源，继承了 IntelliJ IDEA 强大的 JavaScript 功能。

WebStorm 官网下载链接为 http://www.jetbrains.com/webstorm/，通过该链接可以进入 WebStorm 下载界面，如图 2-4 所示。



图 2-4　WebStorm 下载页面

注：官网下载的 WebStorm 可以试用 30 天，试用期过后需要购买。

下载安装后，即可使用 WebStorm。WebStorm 集成了 React 的开发环境，在创建一个 React App 后，WebStorm 会自动下载 React 需要的各种依赖包，无须开发者再次手动配置。该工具占用资源比较多，如果读者不喜欢，后面我们介绍项目时也会给出脚手架创建项目的方式，没有安装这个工具不会受到影响。

2.3 编写第一个 React 应用

WebStorm 准备就绪后，就可以搭建一个简单的 React App 了。打开 WebStorm，选择 Create New Project（创建一个项目），就会看到图 2-5 所示的界面，选择 React App 选项。

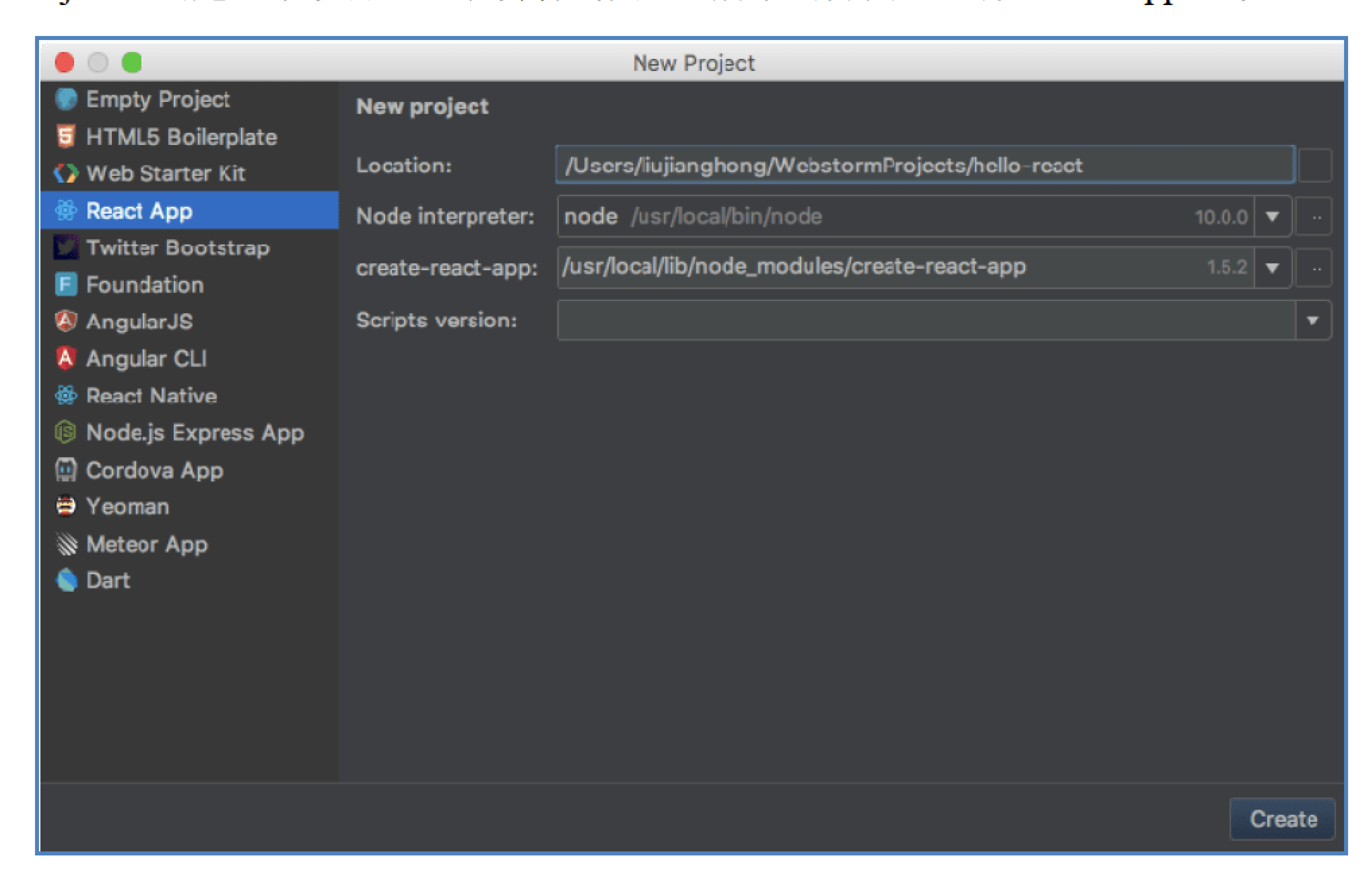


图 2-5　创建一个 React App 工程

其中：

- **Location**：指定项目路径及项目名称。
- **Node interpreter**：指定 Node 路径及显示 Node 版本。
- **create-react-app**：指定 create-react-app 的安装路径及显示版本号。
- **Scripts version**：指定脚本的版本号。



如果是第一次使用 create-react-app，需要用 npm 去安装该脚手架，命令如下：

```
$ npm install -g create-react-app
```

单击 Create 按钮创建项目，这个过程需要 1 分钟左右，这段时间里，create-react-app 会帮

助开发者创建必要的配置文件以及下载 React 项目所需要的各种依赖模块。这个过程的日志会在 WebStorm 中的 Run 栏中显示，内容如下：

```
    Creating a new React app in /Users/liujianghong/WebstormProjects/hello-react.
    Installing packages. This might take a couple of minutes.//这里开始下载依赖模块
    Installing react, react-dom, and react-scripts...
    > fsevents@1.2.3 install /Users/liujianghong/WebstormProjects/hello-react/
node_modules/fsevents
    > node install
    Success:"/Users/liujianghong/WebstormProjects/hello-react/node_modules/fsev
ents/lib/binding/Release/node-v64-darwin-x64/fse.node" is installed via remote
    > spawn-sync@1.0.15 postinstall /Users/liujianghong/WebstormProjects/
hello-react/node_modules/spawn-sync
    > node postinstall
    >uglifyjs-webpack-plugin@0.4.6 postinstall/Users/liujianghong/
WebstormProjects/hello-react/node_modules/uglifyjs-webpack-plugin
    > node lib/post_install.js
    + react-dom@16.3.2
    + react@16.3.2
    + react-scripts@1.1.4
    added 1597 packages in 67.109s  //依赖包下载完成，共花费时间为 67.109s
    Success! Created hello-react at /Users/liujianghong/WebstormProjects/
hello-react
    Inside that directory, you can run several commands:
                                    //提示开发者可以用到以下几个命令
      npm start                     //启动部署到服务
        Starts the development server.
      npm run build                 //把 app 打包到静态资源中
        Bundles the app into static files for production.
      npm test                      //启动测试
        Starts the test runner.
      npm run eject                 //移除项目的单一依赖构建
        Removes this tool and copies build dependencies, configuration files
        and scripts into the app directory. If you do this, you can't go back!
    We suggest that you begin by typing:
                     //提示开发者，可以通过终端进入项目根目录，执行 npm start 命令启动项目
      cd /Users/liujianghong/WebstormProjects/hello-react
      npm start
    Happy hacking!
    Done                    //React 项目构建完成
```

接下来，进入 WebStorm 的 Terminal 模式（在主界面下方单击 Terminal 选项），执行 npm start 命令，项目进行构建后运行，终端提示如图 2-6 所示的信息。

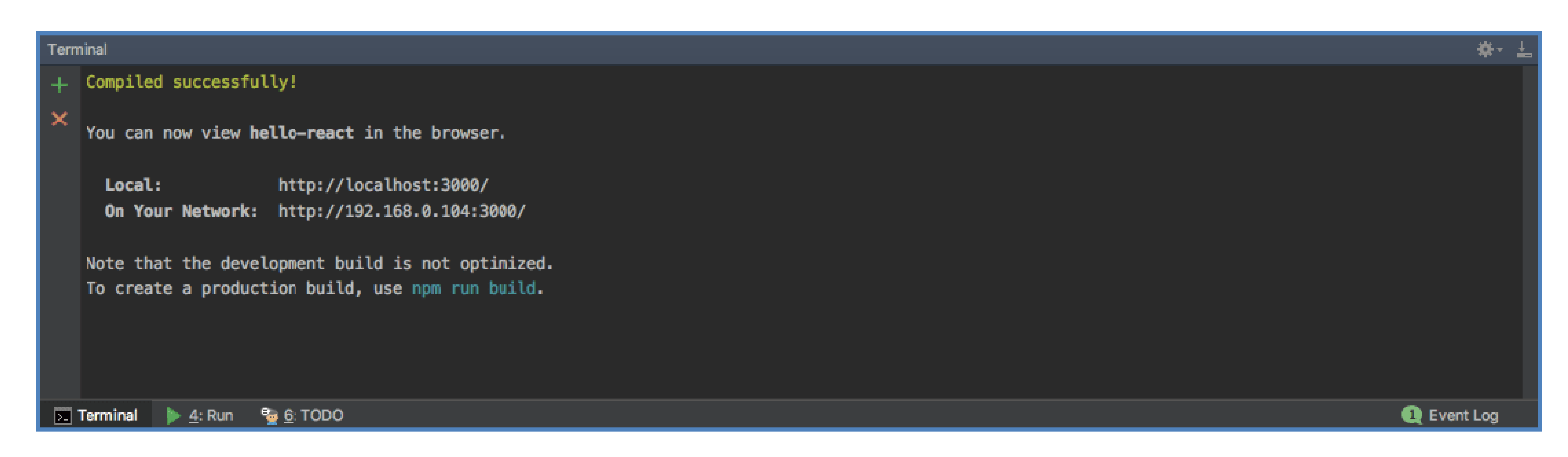


图 2-6　终端提示项目已运行

在终端提示中，项目在浏览器的地址为 http://localhost:3000/ 或者为 http://192.168.0.104:3000/（192.168.0.104 是笔者电脑的 IPv4 地址，在真正搭建项目时，该 IP 改为读者的本地 IP）。create-react-app 在构建项目时，端口号默认设置为 3000。如果开发者想修改端口号，可以在 node_modules/react-scripts/scripts/start.js 文件中修改，该文件配置了项目启动的 IP 以及端口号，代码如下：

```
const DEFAULT_PORT = parseInt(process.env.PORT, 10) || 3000;   //端口号修改处
const HOST = process.env.HOST || '0.0.0.0';
```

至此，React 的第一个项目已经可以访问，打开浏览器，输入地址“http://localhost:3000/”或者“http://192.168.0.104:3000/”即可访问运行效果，如图 2-7 所示。

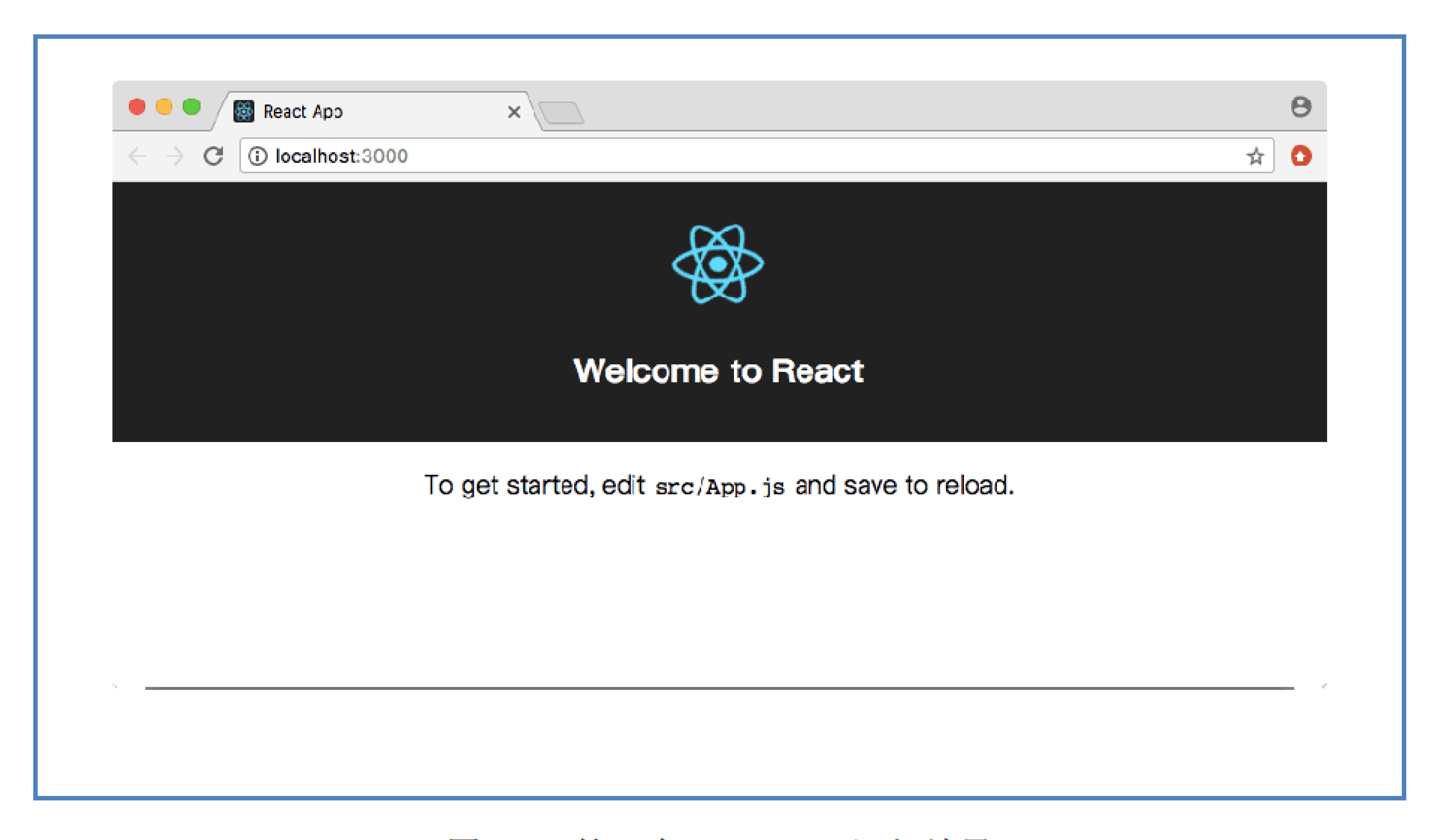


图 2-7　第一个 React App 运行效果

create-react-app 创建项目的默认路径是 C:\Users\机器名\WebstormProjects，默认文件目录如下：

```
├── node_modules
├── README.md
├── package-lock.json
├── package.json
```

```
├── public
|     ├─── favicon.ico
|     ├─── index.html
|     └─── manifest.json
└──── src
        ├─── App.css
        ├─── App.js
        ├─── App.test.js
        ├─── index.css
        ├─── index.js
        ├─── logo.svg
        └─── registerServiceWorker.js
```

其中：

- node_modules：该目录里放的是项目需要的各种依赖模块。
- README.md：该文件会写一些关于项目说明的内容。
- package-lock.json：该文件用来记录当前状态下实际安装的各个包的来源以及版本号。
- package.json：该文件用来定义依赖关系树，以及各个依赖模块的版本范围。
- public：该目录的文件被 index.html 引用。
- src：该目录存放源码以及引用的一些.css、.js 文件，只有该目录下的文件才会被 webpack 识别。



初学者不用害怕这么多文件，本书前几章的示例都会采用直接引入 React 的方法来学习 React 基础，只需要一个 HTML 文件就能搞定。

2.4 与传统 jQuery 对比

jQuery 是由约翰·雷西格（John Resig）在 2006 年 1 月份发布的一套跨浏览器 JavaScript 库，极大地简化了 HTML 与 JavaScript 之间的操作。当时深受前端开发人员的欢迎，jQuery 在操作 DOM、处理事件等方面带来了福音，使得开发效率得到空前提升。

随着时间慢慢推移，人们在开发一些比较复杂的大型项目时，传统的 jQuery 变得越来越难用：一方面是性能问题，由于 jQuery 经常性地操作 DOM 元素，会消耗大量的运行时间；另一方面，用 jQuery 去编写复杂的 DOM 操作，代码会有大量堆积现象，很难维护。当然，jQuery 流行多年，自然有其独特的优点，和当下流行的 React 相比，各有秋色，只是分项目类型而已。

现在要实现一个数值增加功能，思路大概为：在 HTML 中定义两个元素，一个为段落标签，用来显示数值大小；另一个为按钮，用来触发数值增加事件。jQuery 的写法如示例 2-4 所示。

【示例 2-4　jQuery 实现数值增加功能】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>数值增加</title>
    <script src="jquery-3.3.1.min.js"></script>
    <script type="text/javascript">              //jQuery 实现逻辑功能
        $(function () {
            var num = 1;
            $(".myBtn").click(function(){       //获取 DOM 元素，并为其绑定单击事件
                num ++;
                $("p").html(num);
            })
        })
    </script>
</head>
<body>
<div>
    <div>
        <p>1</p>
        <button class="myBtn">增加器</button>
    </div>
</div>
</body>
</html>
```

该示例中用到的两个标签在<body></body>中定义，jQuery 首先利用选择器来获取需要用到的两个 DOM 元素，然后进行逻辑运算之后，再赋值给 DOM 元素。jQuery 更多关注的是实现逻辑过程，然后操作 DOM 改变其状态。

下面用 React 实现同样的功能，如示例 2-5 所示。

【示例 2-5　React 实现数值增加功能】



本书有很多示例只实现了单一功能，为描述方便，使用<script></script>直接引入 React，这样只需要一个 HTML 文件即可，无须配置。待学习到第 8 章 React 架构时，会使用脚手架搭建的整体项目来介绍示例。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>MyButton</title>
```

```
        <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="example"></div>
    <script type="text/babel">
        class AddNumber extends React.Component{
            constructor(props){                     //构造函数，用来初始化 state
                super(props);
                this.state = {number:1};
                this.addNum = this.addNum.bind(this);    //绑定 addNum()方法
            }
            addNum(){                                    //数值增加操作
                this.setState({
                    number:this.state.number+1
                })
            }
            render() {                                   //渲染 DOM 元素
                return (
                    <div>
                        <p>{this.state.number}</p>
                        <button onClick={this.addNum}>增加器</button>
                    </div>
                );
            }
        }
        ReactDOM.render(
            <AddNumber />,
            document.getElementById('example')
        );
    </script>
    </body>
    </html>
```

React 的实现比较整体，定义一个类 AddNumber，其内部既实现了逻辑操作，也渲染了 DOM 元素，并且其状态值用 state 来跟踪。当 state 发生变化时，render()方法才会把最后的结果进行 DOM 渲染。

React 相对 jQuery 有如下几点优势：

- React 的组件化要比 jQuery 的随时操作 DOM 方式更整体，对于开发者来说，React

的这种方式写起来更加舒服，代码写起来会更优雅。

- React 的关注点更多在组件的 state 上，而不用担心组件有没有更新，因为 React 如果认为 state 发生变化，会自动调用 render()方法进行 DOM 渲染。但是 jQuery 需要开发者自己完成这些事情。
- React 中采用了虚拟 DOM 技术，如果 state 中间发生了很多次变化，但是第一次和最后一次的状态是相同的，那 React 会认为该组件状态未更改，不会进行真正的 DOM 渲染，但是 jQuery 不一样，jQuery 需要对每一次的状态变化进行 DOM 操作，这样会浪费页面的很多运行时间，性能相对 React 来说差得多。
- 如果是一个复杂项目，采用 React 的组件化思想，耦合度较低，多人合作开发，各个模块相对独立，后期好维护；用 jQuery 写一些大型复杂的项目，如果没有一个好的架构组织，后期的项目维护会变得越来越难。

2.5 React 调试

调试可以理解为一个寻找程序错误的过程。程序调试是一个开发人员必须具备的能力，调试程序的方法有多种：

- 手动调试。早期 JavaScript 程序的调试，没有辅助工具，开发人员需要手动在程序中输出日志，比如利用 console.log()或者 alert()来进行错误排查。
- 利用工具来调试。随着浏览器的不断改进，各大浏览器提供了 JavaScript 的调试功能，开发者可以利用浏览器提供的工具来进行程序调试。

React 能够发展如此之快，其调试功能强大，也起着举足轻重的作用，本节讲解 React 程序如何进行调试。

2.5.1 安装 Chrome 插件

打开 Chrome 浏览器，输入网址“https://chrome.google.com/webstore/category/extensions”，进入 Chrome 网上应用商店。输入关键字“React Developer Tools”进行搜索，如图 2-8 所示。

单击“添加至 CHROME”按钮，即可安装。然后打开 Chrome 浏览器，从主菜单中选择“更多工具”|“开发者工具”选项（或按 F12 键），如果控制台有 React 选项，表明 React 的调试工具已安装成功，如图 2-9 所示。

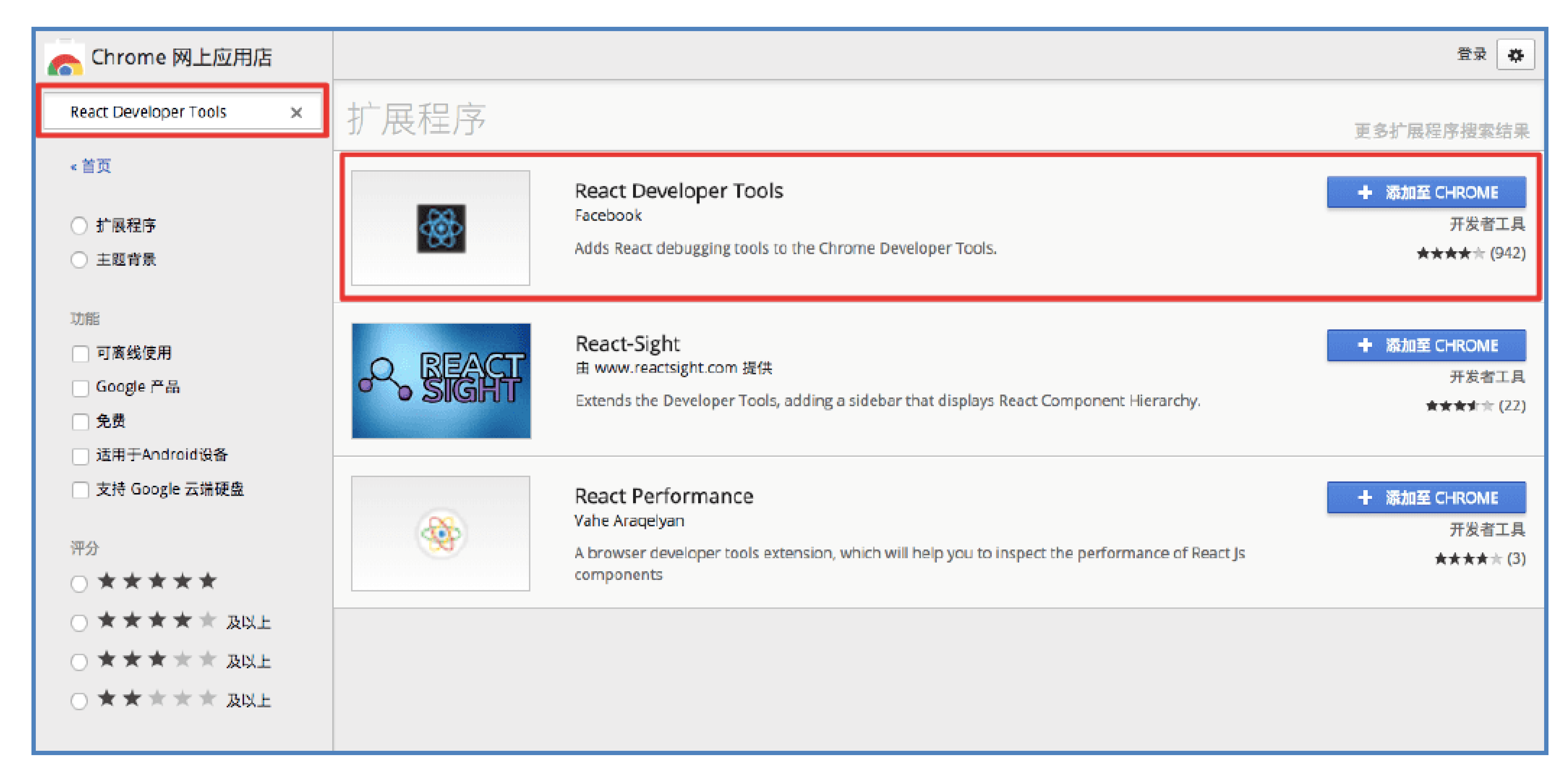


图 2-8　Chrome 网上应用商店中的 React Developer Tools 搜索结果

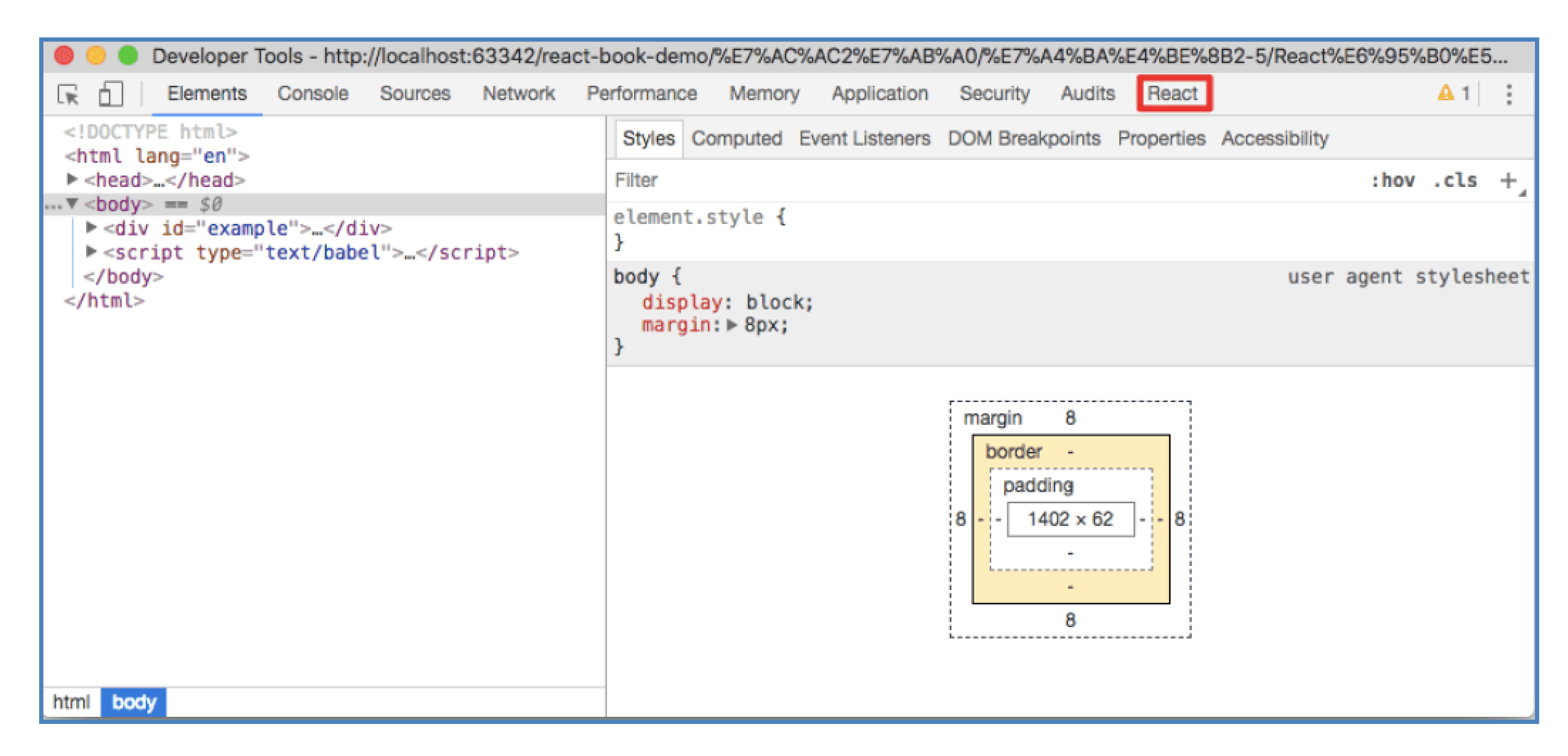


图 2-9　React 调试插件安装成功



国内查看 Chrome 的应用商店有很多限制。如果是苹果用户，建议使用 WiseVPN，笔者就是用的这个 VPN，速度很快，也便宜。如果是 Windows 用户，可以在网上找找有没有更好的选择。

2.5.2　Chrome 插件的使用

这里以 2.4 节中的【示例 2-5】为例，讲解如何利用 React Developer Tools 调试 React 程序。

（1）首先运行程序，并打开 Chrome 浏览器的开发者模式，如图 2-10 所示。

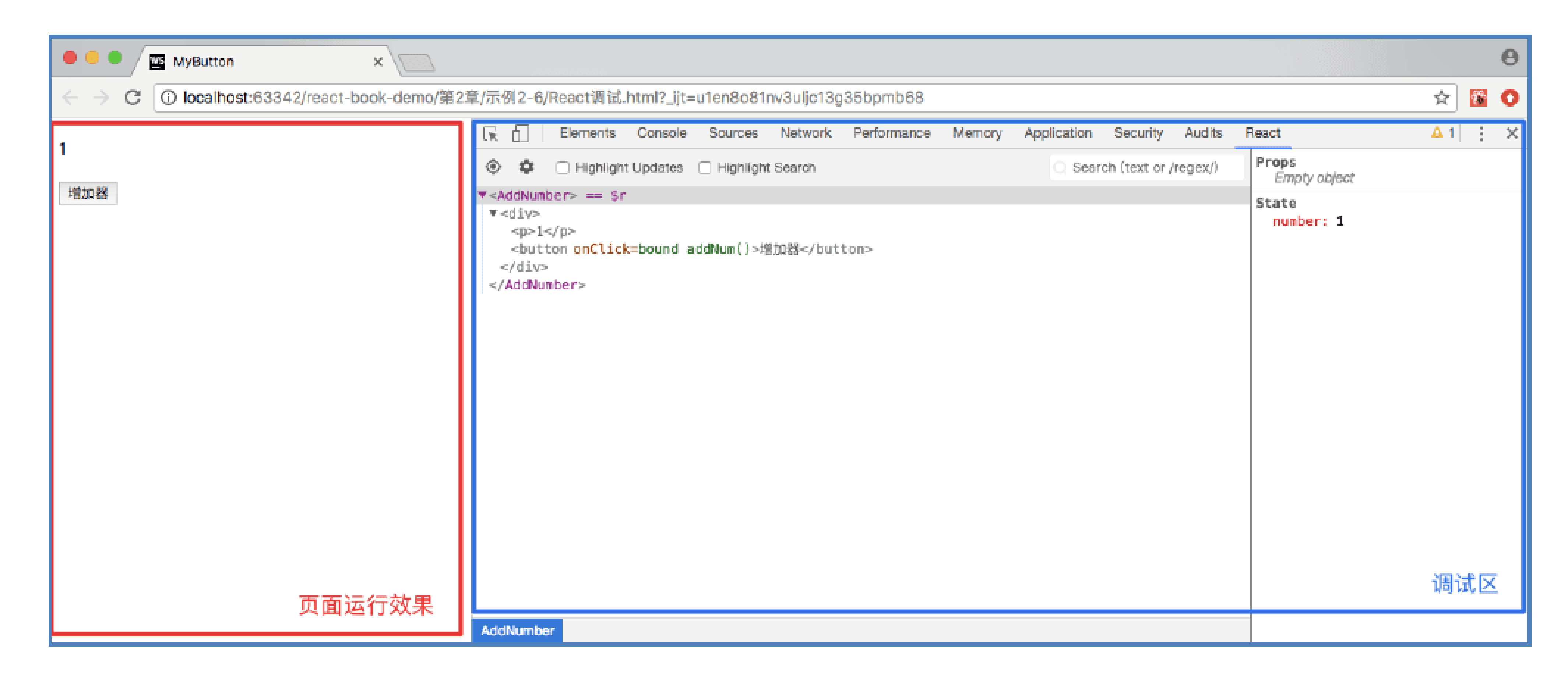


图 2-10 调试界面

（2）在调试区展示的是项目组件，鼠标悬浮在组件之上，单击鼠标右键，会出现 Show AddNumber source 选项（AddNumber 是组件名称），如图 2-11 所示。

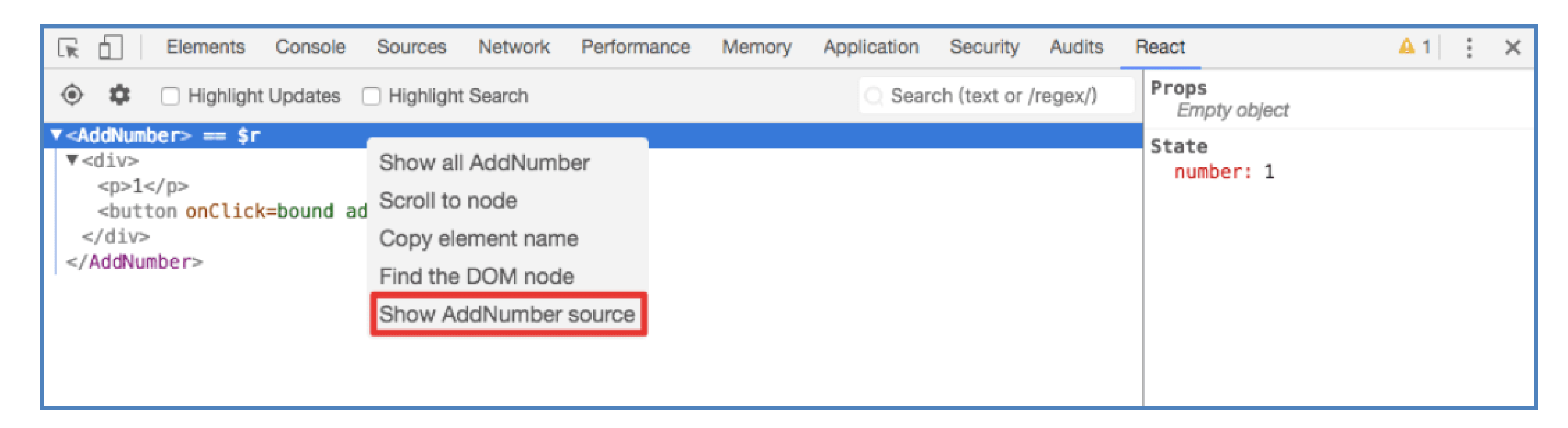


图 2-11 组件源码选项

（3）单击 Show AddNumber source 选项，进入组件源码页面，就可以设置断点进行调试了，如图 2-12 所示。

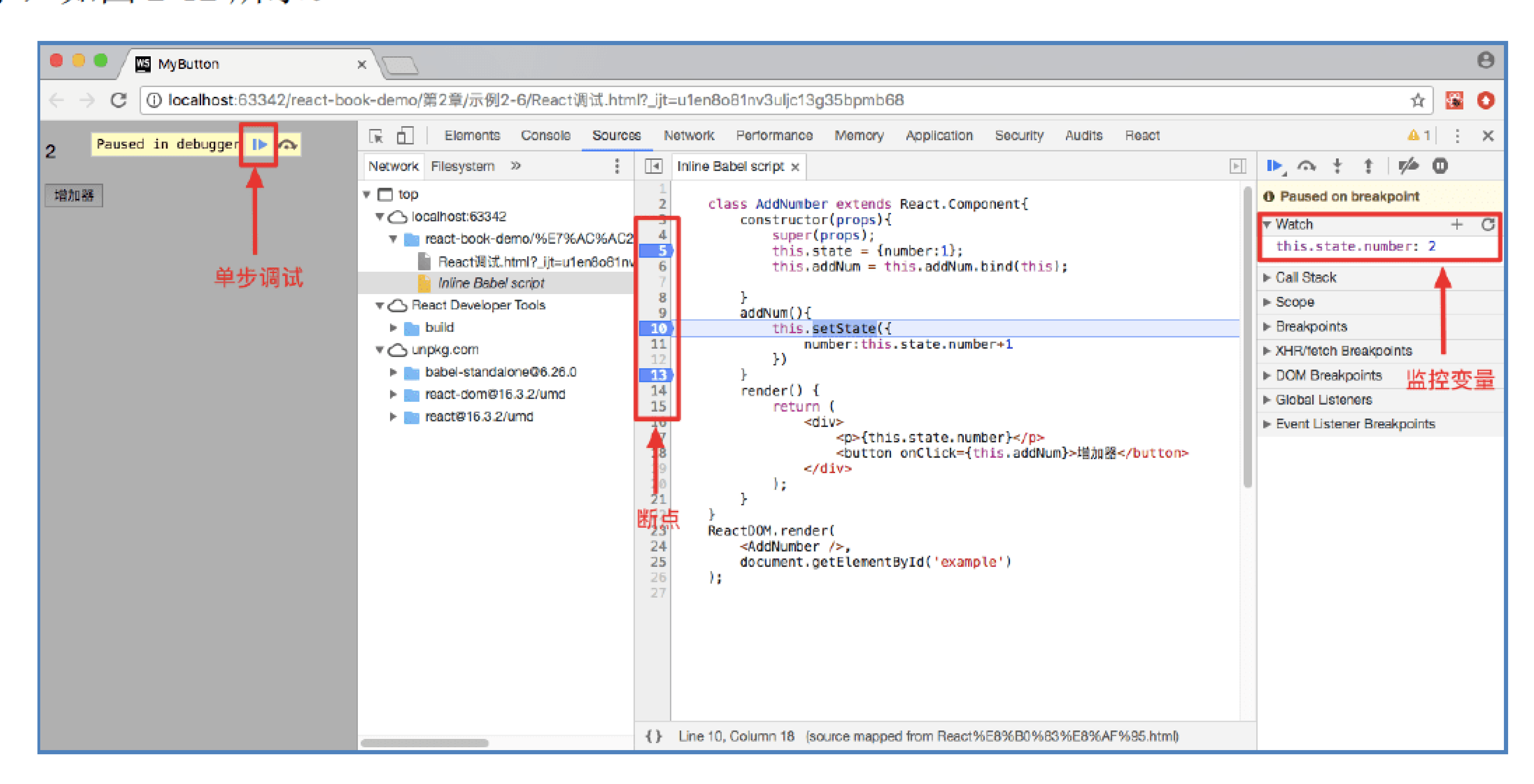


图 2-12 调试示意图

MAC 用户的单步调试快捷键为 fn+F8。Windows 用户直接按 F8 键就是单步调试。其实上述调试方法和以前传统的 JavaScript 调试是一样的。



React Developer Tools 只对 React 程序调试有效，对 React Native 是无效的。

第 3 章

◂React组件▸

React 组件是 React 框架的一个重点，其实 React 诞生的初衷就是为了组件化，也可以理解为实现特定功能的模块化思想。传统的 Web 页面是由 html 基本标签构成的，但在 React 中，构建页面的基本单位是 React 组件。读者在理解 React 组件时，可以将其理解为混合了 JavaScript 具有更好表达能力的 html 元素。在具备 React 组件的熟练编写能力后，读者可以真正体会到高内聚、低耦合、代码高度复用、项目迭代快等诸多优点。

3.1 理解组件化思想

React 中的组件化思想，可以理解为把具有独立功能的 UI 部分进行了封装。

对 MVC 开发模式熟悉的读者应该知道，MVC 模式是将模型、视图、控制器进行分离，从而实现表现层、数据层及控制层的独立。其中，以往的开发者对表现层进行松耦合优化时，基本是从技术的角度对 UI 进行分离的。而 React 提供了一个新的思路，从功能的角度将 UI 封装成不同组件，整个页面的组成都是通过小组件构建成大组件的方式来实现。这样组件化思想就体现出一些独特的优点：

- 可组合性：定义了一个 UI 组件后，可以和其他组件进行并列或者嵌套使用，多个小组件还可以构建一个复杂组件，一个复杂的组件也可以分解成多个功能简单的小组件。
- 可重用性：定义后的组件功能是相对独立的，在不同的 UI 场景中，可以重复使用。
- 可维护性：每个组件的实现逻辑都仅限于自身，不涉及其他组件，这样的可维护性较高。

3.2 组件之间的通信

在 React 进行项目开发中，难免会进行组件之间的信息传递操作，即组件间的通信。通信可以简单地理解为组件之间的数据传递，如父子组件之间的通信、同级组件之间的通信等。

在学习组件通信之前，读者应该先对 React 中的 state 和 props 有所掌握。state 和 props 是

React 中的两种数据方式，无论是 state 数据还是 props 数据只要发生数据改变，就会重新渲染组件。那么本节就来讲述 state 和 props 的使用方法。

3.2.1　props

props 是 properties 的简称，范围为属性。在 React 中的组件都是相对独立的，一个组件可以定义接收外界的 props，在某种意义上就可以理解为组件定义了一个对外的接口。组件自身相当于一个函数，props 可以作为一个参数传入，旨在将任意类型的数据传给组件。

props 的使用方法如示例 3-1 所示。

【示例 3-1　props 简单使用】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>props 简单使用</title>
    <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script
src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>

</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class SayName extends React.Component{
        render() {                          //DOM 渲染
            return (
                <h1>Hello {this.props.name}</h1>
            );
        }
    }
    ReactDOM.render(<SayName name="天虹" />, document.getElementById('root'));
</script>
</body>
</html>
```

在上面的例子中，先定义一个名为 SayName 的组件（注意：在 React 中定义组件一定要用大驼峰法，首字母要大写），在 render()中返回一个<h1>的 DOM 节点，整体渲染<SayName>这个组件的时候，用 this.props.name 来获取 name 属性。当然 props 也可以在挂载组件的时候

为其设置初始值，如示例 3-2 所示。

【示例 3-2　初始化 props】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>初始化 props</title>
    <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script
src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class SayName extends React.Component{
        static defaultProps = {              //初始化 props
            name:'天虹'
        }
        render() {
            return (
                <h1>Hello {this.props.name}</h1>
            );
        }
    }
    ReactDOM.render(<SayName />, document.getElementById('root'));
</script>
</body>
</html>
```

props 一般不允许更改，所以这里用 static 关键字来定义默认的 props 值。在 ES5 中初始化 props 使用 getDefaultProps()方法来实现，在 ES6 中统一使用 static 类型来定义。



props 的属性值不允许组件自己修改，如果需要修改 props 值，请使用 state（后面会介绍）。

3.2.2　state

state 为状态之意。组件在 React 中可以理解为是一个状态机，组件的状态就是用 state 来记录的。相对 props 来说，state 是用在组件内部并且是可以修改的。

在 state 的操作中，基本操作包括初始化、读取和更新。下面通过一个示例来介绍 state 的基本操作，实现改变 DOM 元素颜色。

【示例 3-3　利用 state 改变 DOM 元素颜色】

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>利用 state 改变 DOM 元素颜色</title>
   <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
   <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
   <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
   class ChangeColor extends React.Component{
      constructor(props) {
         super(props);
         this.state = {isRed: true};          //在构造函数中对 state 进行初始化
         this.handleClick = this.handleClick.bind(this);
                                    //在 ES6 中，this 需要在构造函数中绑定后才能生效
      }
       handleClick(){
         this.setState((prevState, props) => ({
            isRed: !prevState.isRed
         }));
      }
      render() {
         var redStyle={
            color:"red"
         }
         var blueStyle={
            color:"blue"
         }
         return (
            <div>
               <h1 style={this.state.isRed ? redStyle:blueStyle}>天虹</h1>
               <button onClick={this.handleClick}>点击改变颜色</button>
            </div>
```

```
            );
        } }
        ReactDOM.render(<ChangeColor />, document.getElementById('root'));
</script>
</body>
</html>
```

在上面的例子中，state 的初始化放在 constructor 的构造函数中，该例子中 isRed 就是一个组件的 state，初始化为 true。接下来在组件渲染中，即调用 rend 函数时，<h1>中的 this.state.isRed 值为 true，则 style 为 redStyle，即红色。在<button>中绑定 handleClick 方法，利用 setState 来改变 state 中的 isRed 值。setState 方法有两个参数，官方给出的说明如下：

```
void setState(
     function|object nextState,
     [function callback]
   )
```

第一个参数表示的是要改变的 state 对象，第二个参数是一个回调函数，这个回调函数是在 setState 的异步操作执行完成并且组件已经渲染后执行的。所以，可以通过该方法获取之前的状态值 prevState。该示例就是通过 setState 获取之前的状态值 prevState，对其进行更新操作，从而重新渲染组件。

3.2.3　父子组件通信

在介绍组件通信之前，先引入一个概念：数据流。在 React 中，数据流是单向的，通过 props 从父节点传递到子节点，如果父节点的 props 发生改变，则 React 会遍历整棵组件树，从而渲染用到这个 props 的所有组件。

父组件更新子组件，可以直接通过 props 进行信息传递，实现更新操作，如示例 3-4 所示。

【示例 3-4　子父组件通信】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>子父组件通信</title>
    <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script
src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>
```

```
<div id="root"></div>
<script type="text/babel">
    class Child extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        render(){
            return (
                <div>
                    {this.props.text}
                </div>
            )
        }
    }
    class Father extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        refreshChild(){
            return (e)=>{
                this.setState({
                    childText: "父组件更新子组件成功",
                })
            }
        }
        render(){
            return (
                <div>
                    <button onClick={this.refreshChild()} >
                        父组件更新子组件
                    </button>
                    <Child text={this.state.childText || "子组件更新前"} />
                </div>
            )
        }
    }
    ReactDOM.render(<Father />, document.getElementById('root'));
</script>
</body>
</html>
```

在父组件<Father />中，dom 节点<button>绑定 refreshChild()方法，在 refreshChild()方法中

直接修改 state 值，传给子组件即可。

子组件更新父组件，需要父组件传一个回调函数给子组件，然后子组件调用，就可以触发父组件更新了，如示例 3-5 所示。

【示例 3-5　父子组件通信】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>父子组件通信</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class Child extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        render(){
            return (
                <div>
                    <button onClick={this.props.refreshParent}>
                        更新父组件
                    </button>
                </div>
            )
        }
    }
    class Father extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        refreshParent(){
            this.setState({
                parentText: "子组件更新父组件成功",
```

```
                })
            }
            render(){
                return (
                    <div>
                        <Child refreshParent={this.refreshParent.bind(this)} />
                        {this.state.parentText || "父组件更新前"}
                        </div>
                )
            }
        }
        ReactDOM.render(<Father />, document.getElementById('root'));
    </script>
    </body>
    </html>
```

子组件<Child />调用父组件的 refreshParent()方法，触发后修改 state 值，从而更新父组件内容。

3.2.4　同级组件通信

所谓同级组件，即组件们有一个共同的祖先，但其各自的辈分一样。同级组件之间的通信有两种方式：

- 第一种：组件与组件之间有一个共同的父组件，可以通过父组件来通信，即一个子组件可以通过父组件的回调函数来改变 props，从而改变另一个子组件的 props。
- 第二种：组件与组件之间有共同的祖先，但不一定是亲兄弟，这样如果通过父组件一级一级去调用，效率会很差。React 提供了一种上下文方式，允许子组件可以直接访问祖先组件的属性和方法，从而使效率得到很大的提升。

兄弟组件通信，如示例 3-6 所示。

【示例 3-6　兄弟组件通信】

```
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>兄弟组件通信</title>
        <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
```

```
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class Brother1 extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        render(){
            return (
                <div>
                    <button onClick={this.props.refresh}>
                        更新兄弟组件
                    </button>
                </div>
            )
        }
    }
    class Brother2 extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        render(){
            return (
                <div>
                    {this.props.text || "兄弟组件未更新"}
                    </div>
            )
        }
    }
    class Father extends React.Component{
        constructor(props){
            super(props);
            this.state = {}
        }
        refresh(){
            return (e)=>{
                this.setState({
                    text: "兄弟组件通信成功",
                })
            }
```

```
            }
            render(){
                return (
                    <div>
                        <h2>兄弟组件沟通</h2>
                        <Brother1 refresh={this.refresh()}/>
                        <Brother2 text={this.state.text}/>
                    </div>
                )
            }
        }
        ReactDOM.render(<Father />, document.getElementById('root'));
    </script>
    </body>
    </html>
```

<Brother1/>和<Brother2/>是两个兄弟组件，二者有一个共同的父组件，第一个子组件<Brother1/>可以通过父组件<Father/>的回调方法修改 state，当父组件的 state 有了改动之后，其改动的 state 会传给其所有子组件后重新渲染，这样就达到了兄弟节点之间的通信。

3.3 组件生命周期

所谓的生命周期，可以用有生命的人体来表达这个意思。从出生到成长，最后到死亡，这个过程的时间可以理解为生命周期。React 的生命周期同理也是这么一个过程。在 React 的工作中，生命周期也一直存在于工作过程中。React 的生命周期严格分为三个阶段：挂载期（也叫实例化期）、更新期（也叫存在期）、卸载期（也叫销毁期）。在每个周期中，React 都提供了一些钩子函数，读者可以依据这些钩子函数很好地理解 React 的生命周期是一个怎样的过程。React 的生命周期描述如下：

- 挂载期：一个组件实例初次被创建的过程。
- 更新期：组件在创建后再次渲染的过程。
- 卸载期：组件在使用完后被销毁的过程。

3.3.1 组件的挂载

组件在首次创建后，进行第一次的渲染称为挂载期。挂载期有一些方法会被依次触发，列举如下：

- constructor（构造函数，初始化状态值）
- getInitialState（设置状态机）

- getDefaultProps（获取默认的 props）
- componentWillMount（首次渲染前执行）
- render（渲染组件）
- componentDIdMount（render 渲染后执行的操作）

这里用一个示例来直观感受一下组件挂载这个阶段的执行过程，如示例 3-7 所示。

【示例 3-7　组件挂载】

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>组件挂载</title>
   <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
   <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
   <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
   class HelloWorld extends React.Component {
      constructor(props) {
         super(props);
         console.log("1. 构造函数")
         this.state = {};
         console.log("2. 设置状态机")
      }
      static defaultProps={
         name:"React"
      }
      componentWillMount(){
         console.log('3. componentWillMout 完成首次渲染前调用')
      }
      render () {
         console.log('4. 组件进行渲染');
         return(
            <div>
               <div>{this.props.name}</div>
            </div>
         )
```

```
        }
        componentDidMount() {
            console.log('5. componentDidMount render 渲染后的操作' )
        } }
         ReactDOM.render(<HelloWorld />, document.getElementById('root'));

</script>
</body>
</html>
```

这里用 ES6 的标准来呈现组件的挂载过程，在组件创建的时候会按照上述代码中的 console 依次输出相关内容。在 ES6 中是不使用 getInitialState 和 getDefaultProps 两个方法的，这是 ES5 的方法。读者注意，ES5 中 React.createClass 渐渐要被 FaceBook 官方废弃，建议以后的 React 写法，最好用 ES6 标准来写。设置 state 的初始值以及获取 props，已经在上述示例中实现：设置 state 初始值在构造函数中用 this.state 来处理，获取 props 用 defaultProps 来处理。

另外还需要注意，在 React 组件中，render 方法必须实现，如果没有实现会报错，其他的方法可以不实现，因为除了 render 方法外，其他方法在父类 Component 中都有默认实现。

上述代码在浏览器上的效果如图 3-1 所示。

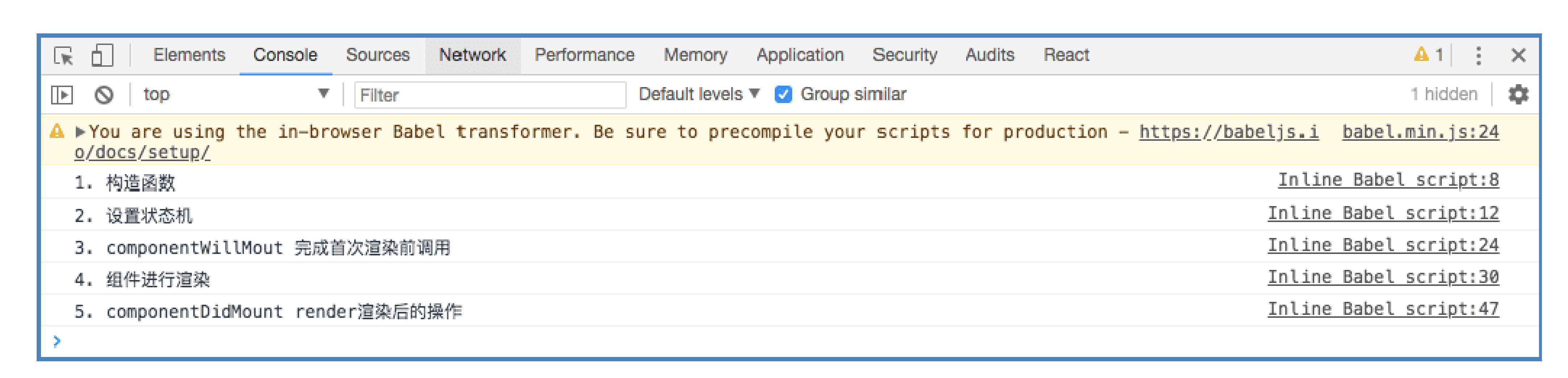


图 3-1　组件挂载中的函数执行结果

3.3.2　组件的更新

组件更新，指的是在组件初次渲染后，进行了组件状态的改变。在实际项目中，组件更新是经常性操作。React 在生命周期中的更新过程包括以下几个方法：

- componentWillReceiveProps：当父组件更新子组件的 state 时，该方法会被调用。
- shouldComponentUpdate：该方法决定组件 state 或者 props 的改变是否需要重新渲染组件。
- componentWillUpdate：在组件接受新的 props 或者 state 时，即将进行重新渲染前调用该方法，和 componentWillMount 方法类似。
- componentDidUpdate：在组件重新渲染后调用该方法，和 componentDidMount 方法类似。

下面用一个具体示例来更好地理解组件更新过程。

【示例 3-8　组件更新】

```
<!DOCTYPE html>
```

```
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>组件更新</title>
        <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class HelloWorldFather extends React.Component{  //父组件
            constructor(props) {
                super(props);
                this.updateChildProps = this.updateChildProps.bind(this);
                this.state = {  //初始化父组件 state
                    name:"React"
                }
            }
            updateChildProps(){  //更新父组件 state
                this.setState({
                    name:"Vue"
                })
            }
            render(){
                return(
                    <div>
                        <HelloWorld name={this.state.name}></HelloWorld>
                                                  //父组件的 state 传递给子组件
                        <button onClick={this.updateChildProps}>更新子组件 props</button>
                    </div>
                )
            }

        }
        class HelloWorld extends React.Component {
            constructor(props) {
                super(props);
                console.log("1. 构造函数")
                console.log("2. 设置状态机")
```

```
        }

        componentWillMount(){
            console.log('3. componentWillMout 完成首次渲染前调用')
        }

        componentWillReceiveProps(){
            console.log("6. 父组件更新子组件 props 时，调用该方法")
        }
        shouldComponentUpdate(){
            console.log("7. 决定组件 props 或者 state 的改变是否需要进行重新渲染")
            return true;
        }
        componentWillUpdate(){
            console.log("8. 当接收到新的 props 或 state 时，调用该方法")
        }
        render () {
            console.log('4. 组件进行渲染');
            return(
                <div>
                    <div>{this.props.name}</div>
                </div>
            )
        }
        componentDidMount() {
            console.log('5. componentDidMount render 渲染后的操作' )
        }
        componentDidUpdate(){
            console.log("9. 组件重新被渲染后，调用该方法")
        }
    }
     ReactDOM.render(<HelloWorldFather />, document.getElementById('root'));
</script>
</body>
</html>
```

该示例是建立在组件挂载示例基础上的组件更新过程，这样读者对生命周期的整个过程会有更好的理解。上述示例描述了两个组件，父组件<HelloWorldFather/>和子组件<HelloWorld/>。子组件的 state 由父组件来更新。整体的实现功能为，子组件首次渲染，属性 name 值为“React”，当父组件 name 值改为“Vue”时，子组件的 state 随之发生改变，进而触发组件生命周期中更新的相关操作。运行效果如图 3-2 所示。

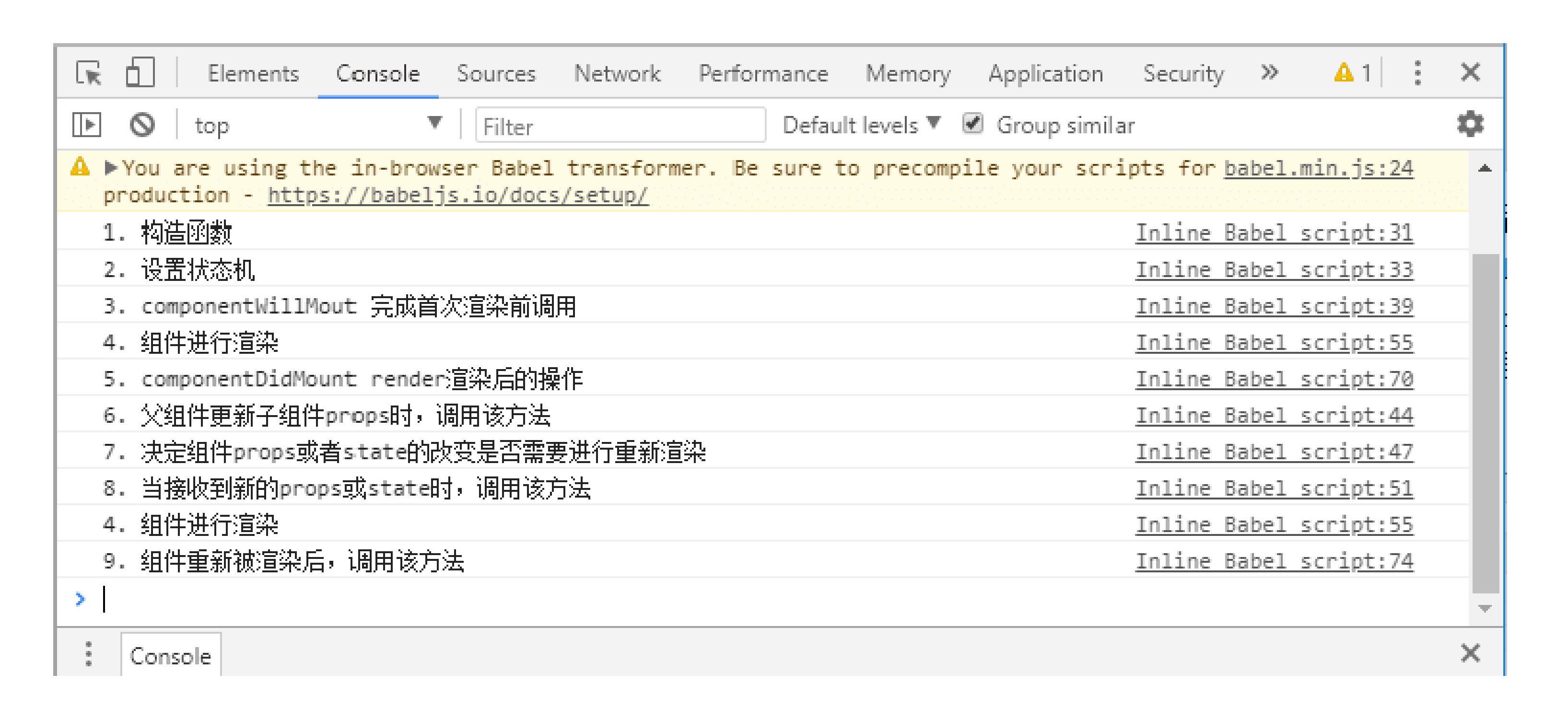


图 3-2　组件更新运行效果

从第 6 步到第 9 步为组件更新过程。



在组件更新过程中，需要注意 shouldComponentUpdate()方法，如果该方法返回值为 false 时，组件将不进行重新渲染。该方法如果能用的恰到好处，就能够在 React 性能优化方面起到一定作用。虽然说 React 的性能已经可以了，但是减少没有必要的渲染依然可以进一步优化性能。

3.3.3　组件的卸载

生命周期的最后一个过程为组件卸载期，也称为组件销毁期。该过程主要涉及一个方法，即 componentWillUnmount，当组件从 DOM 树删除的时候调用该方法。

这里以示例 3-8 为基础，通过添加组件销毁的方法来理解生命周期中的最后一个组件卸载环节，如示例 3-9 所示。

【示例 3-9　组件卸载】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>组件卸载</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class HelloWorldFather extends React.Component{
```

```
    constructor(props) {
        super(props);
        this.updateChildProps = this.updateChildProps.bind(this);
        this.state = {
            name:"React"
        }
    }
    updateChildProps(){
        this.setState({
            name:"Vue"
        })
    }
    render(){
        return(
            <div>
                <HelloWorld name={this.state.name}></HelloWorld>
                <button onClick={this.updateChildProps}>更新子组件 props</button>
            </div>
        )
    }
}
class HelloWorld extends React.Component {
    constructor(props) {
        super(props);
        console.log("1. 构造函数")
        console.log("2. 设置状态机")
    }

    componentWillMount(){
        console.log('3. componentWillMout 完成首次渲染前调用')
    }
    componentWillReceiveProps(){
        console.log("6. 父组件更新子组件 props 时，调用该方法")
    }
    shouldComponentUpdate(){
        console.log("7. 决定组件 props 或者 state 的改变是否需要进行重新渲染")
        return true;
    }
    componentWillUpdate(){
        console.log("8. 当接收到新的 props 或 state 时，调用该方法")
    }
    delComponent(){   //添加卸载方法
        ReactDOM.unmountComponentAtNode(document.getElementById('root'));
    }
```

```
        render () {
            console.log('4. 组件进行渲染');
            return(
                <div>
                    <div>{this.props.name}</div>
                    <button onClick={this.delComponent}>卸载组件</button>
                         //声明卸载按钮
                </div>
            )
        }
        componentDidMount() {
            console.log('5. componentDidMount render 渲染后的操作' )
        }
        componentDidUpdate(){
            console.log("9. 组件重新被渲染后，调用该方法")
        }
        componentWillUnmount(){  //组件卸载后执行
            console.log("10. 组件已被销毁")
        }
    }
    ReactDOM.render(<HelloWorldFather />, document.getElementById('root'));
</script>
</body>
</html>
```

上述示例在示例 3-8 的基础上通过添加卸载事件来卸载组件，这里用到的方法为 unmountComponentAtNode()，参数为 DOM 的 ID，当组件卸载后，调用生命周期中的卸载钩子方法 componentWillUnmount()。效果如图 3-3 所示。

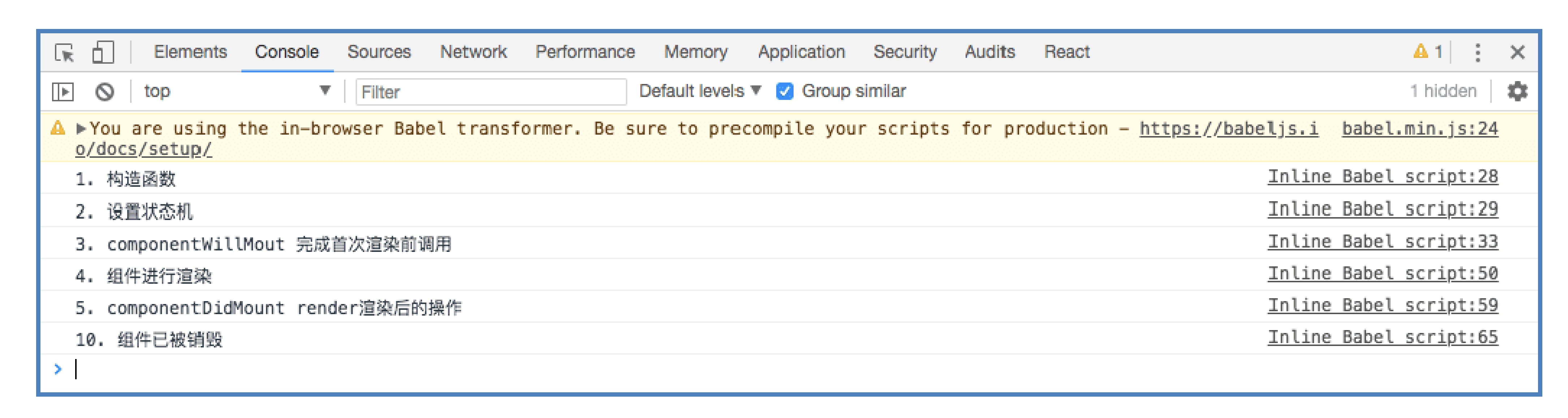


图 3-3　组件卸载效果

3.3.4　总览组件生命周期

React 组件生命周期经历三个阶段：组件挂载期、组件更新期、组件卸载期。整个过程大致可以用图 3-4 来描述。

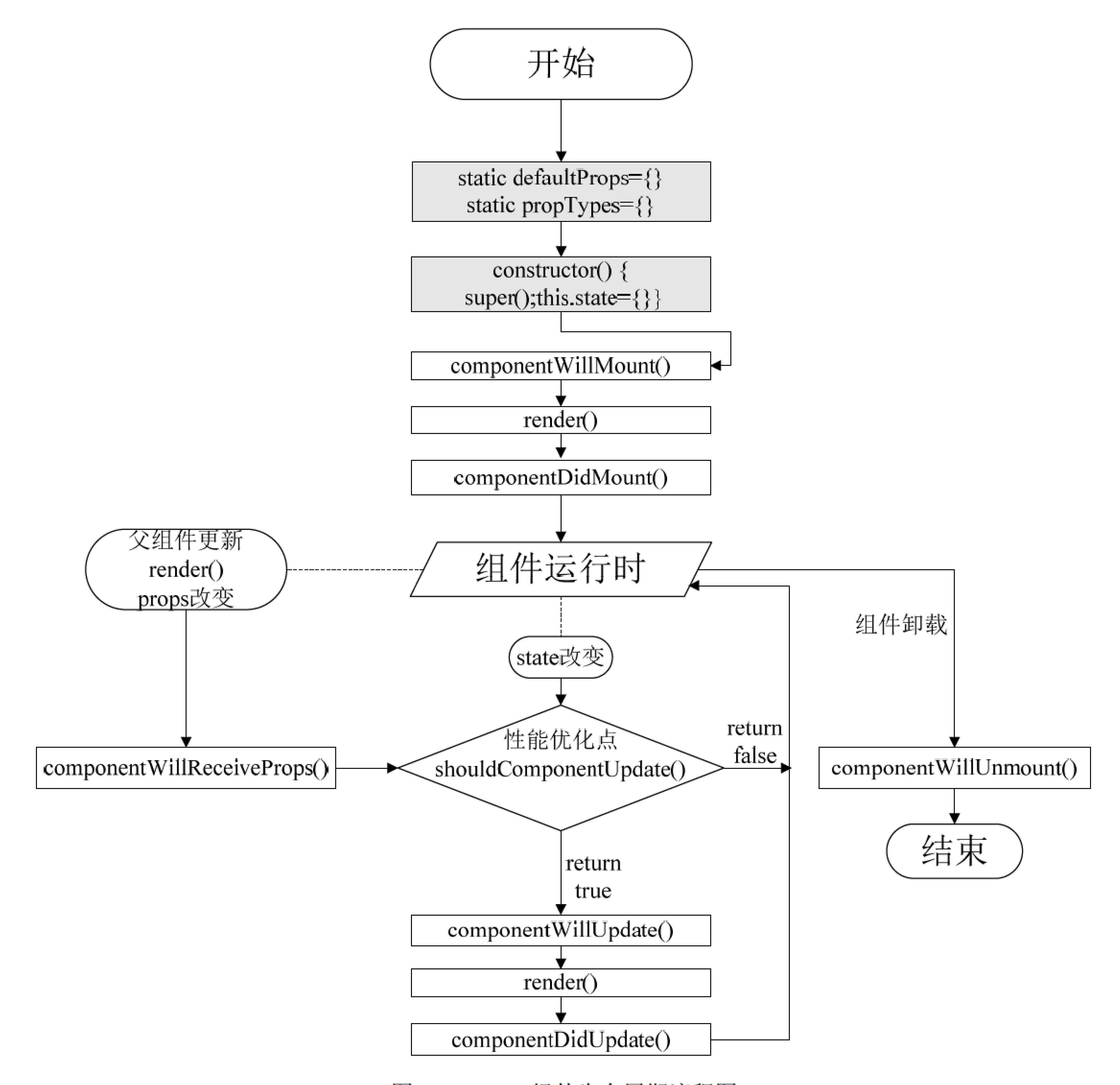


图 3-4　React 组件生命周期流程图

React 整个生命周期提供了完整的钩子方法，这些方法伴随着整个组件从创建到销毁。React 生命周期中主要做的事情是围绕组件 state 和 props 的状态改变而产生的一系列操作。读者在实际项目开发中，如果能真正理解 React 的生命周期，对编码的整体性会有很好的把握。

第 4 章

漫谈React事件系统

一个前端项目，基本可以分为两大部分：一部分为视觉呈现，另一部分为功能使用。谈到用户的功能操作，多数情况都件随着事件发生，比如用鼠标单击一个按钮发送消息、将鼠标悬浮在图片上显示阴影效果等。React 提供了一套非常完善的事件系统，其基于 DOM 事件体系之上，做了很多性能优化以及浏览器兼容等改善，给开发者带来了更多便利。本章主要讲解 React 的事件系统。

4.1 JavaScript 事件机制

所谓事件，简单可以理解为要做一件什么事情。事件系统，是整个事件的所有处理系统。一般而言，事件系统大概包含 3 个重要因素：

- 事件源：动作发生的初始点。
- 事件对象：保存事件状态。
- 事件处理：要做一件什么事情。

JavaScript 是一门单线程非阻塞的脚本语言，非阻塞主要体现在异步功能上。读者应该明白，在前端项目中会有大量的 DOM 操作或者 I/O 事件等，为了避免出现阻塞现象，JavaScript 采用了事件循环机制来解决这一问题。所谓循环机制，即 JavaScript 引擎提供了一个执行栈和一个事件队列。当一段 JavaScript 代码执行时，同步代码会被加入到执行栈中，然后依次执行，如果有异步代码，则异步事件会被加入到事件队列里，直到执行栈中的所有方法执行完毕后，再从事件队列里面依次取出异步事件，放到执行栈中执行。这里用下面的一个示例来理解一下事件循环原理。

【示例 4-1　事件循环】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>事件循环</title>
```

```
    <script type="text/javascript">
        function fn1() {
            console.log("1. 主线程执行")
        }
        function fn2() {
            setTimeout(function () {
                console.log("2. 先放入事件队列，等执行栈全部执行完后，执行该方法")
            })
        }
        function fn3() {
            console.log("3. 主线程执行")
        }
        fn1();
        fn2();
        fn3();
    </script>
</head>
<body>
</body>
</html>
```

运行结果如图 4-1 所示。

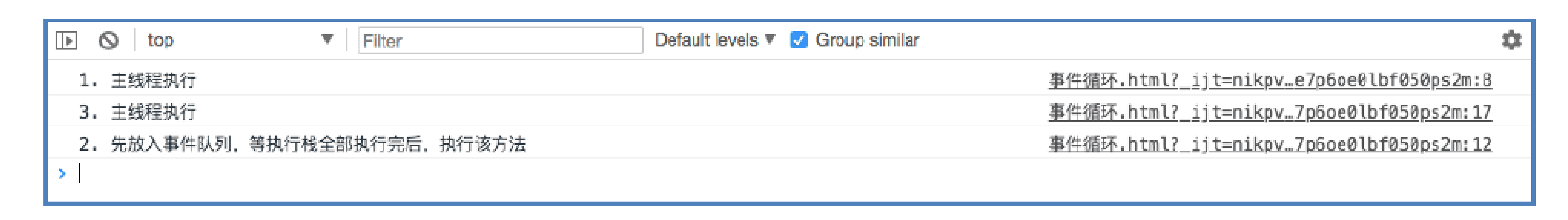


图 4-1　事件循环示例执行结果

通过运行结果可以看到，上述示例执行时，先执行的 fn1()和 fn3()方法，最后执行 fn2()方法。原因是这样的，fn1()和 fn3()方法为同步方法，fn2()为异步方法，当 fn1()方法在执行栈运行完成后看到 fn2()时，识别出这是一个异步方法后将其放入事件队列中等待，接着把 fn3()方法放入执行栈中运行，等 fn3()方法运行完后，发现执行栈已经清空，再去事件队列中寻找还在等待的 fn2()方法，所以实际的运行顺序为 fn1()→fn3()→fn2()。事件循环原理可以用图 4-2 表示。

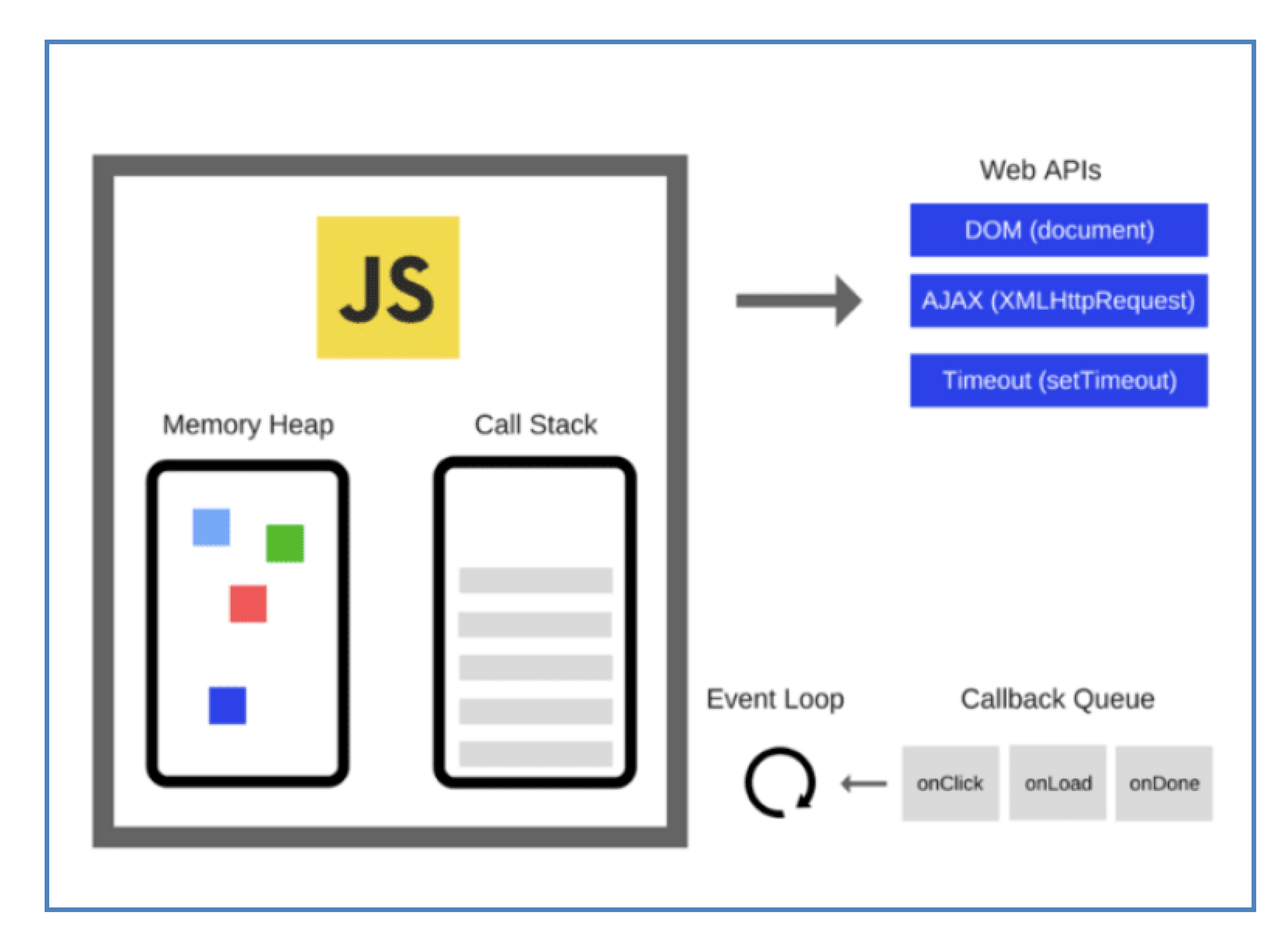


图 4-2　JavaScript 事件循环示意图

在图 4-2 中，Web APIs 为一些异步操作，当 JavaScript 识别出异步方法时，会先将其放入 Callback Queue（这个事件队列中多数为回调函数，所以也称为回调队列）中，Call Stack 中会先执行一些同步操作，等到 Call Stack 中的所有方法都执行完成后，Callback Queue 中等待的方法才会放入执行栈中执行。Memory Heap 中一般存放一些对象，担当一个内存区域的角色。

JavaScript 事件的触发大概分为三个阶段：

- 事件捕获阶段：事件从文档的根节点出发，向其子节点延伸，遇到相同注册事件立即触发，直到目标节点为止。
- 事件处理阶段：事件到达目标节点，触发事件。
- 事件冒泡阶段：事件离开目标节点返回到文档根节点，并在路途上遇到相同注册事件再次触发。

下面通过一个示例来理解这三个阶段。

【示例 4-2　JavaScript 事件三阶段】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JavaScript 事件三阶段</title>
</head>
<body>
<div id="myDivFather">
    父节点
    <div id="myDivSon">
        子节点
```

```
        <div id="myDivGrandson">
            孙节点
        </div>
    </div>
</div>
<script type="text/javascript">
    var f = document.getElementById('myDivFather');
    var s = document.getElementById("myDivSon");
    var g = document.getElementById("myDivGrandson")
    f.addEventListener("click",function () {  //注册单击事件
        console.log("1. father  向下捕获阶段");
    },true);
    s.addEventListener('click',function () {
        console.log("2. son  向下捕获阶段")
    },true);
    g.addEventListener('click',function () {
        console.log("3. grandson  向下捕获阶段");
    },true);
    f.addEventListener('click',function () {
        console.log("4. father  向上冒泡阶段")
    },false);
    s.addEventListener('click',function () {
        console.log("5. son  向上冒泡阶段")
    },false);
    g.addEventListener('click',function () {
        console.log("6. grandson  向上冒泡阶段");
    },false);
</script>
</body>
</html>
```

addEventListener 方法的最后一个参数可以指定事件为捕获还是冒泡，即：

```
obj.addEventListener("click", func, true);   // 捕获方式
obj.addEventListener("click", func, false);   // 冒泡方式
```

第一个参数为事件类型；第二个参数为回调函数，执行响应事件；第三个参数用来指明是捕获事件还是冒泡事件。在上述示例中，有三个节点，即父、子、孙节点，都注册了捕获方式和冒泡方式，当单击孙节点时，事件先到父节点，然后到子节点，最后到孙节点，之后再进行冒泡，返回到子节点，再返回到父节点。单击孙节点时，程序运行效果如图 4-3 所示。

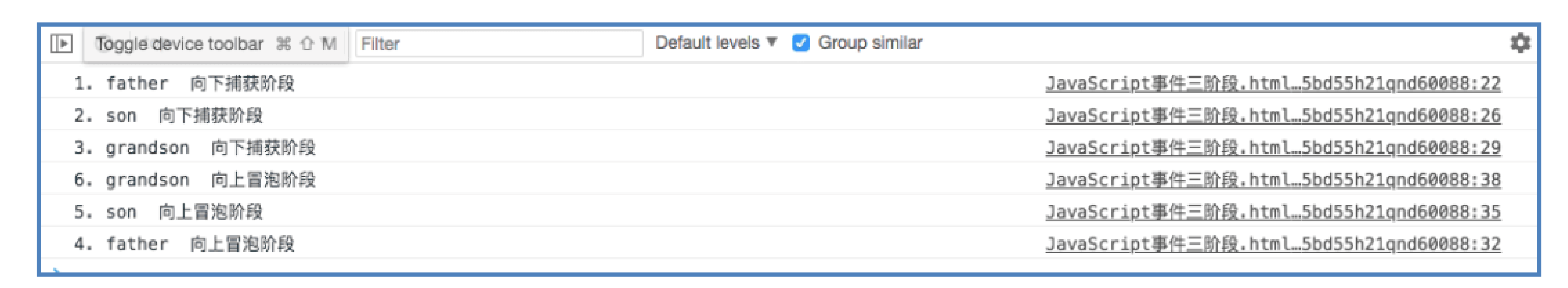


图 4-3　事件捕获和事件冒泡运行结果

JavaScript 事件传递的三个阶段可以用图 4-4 来理解。

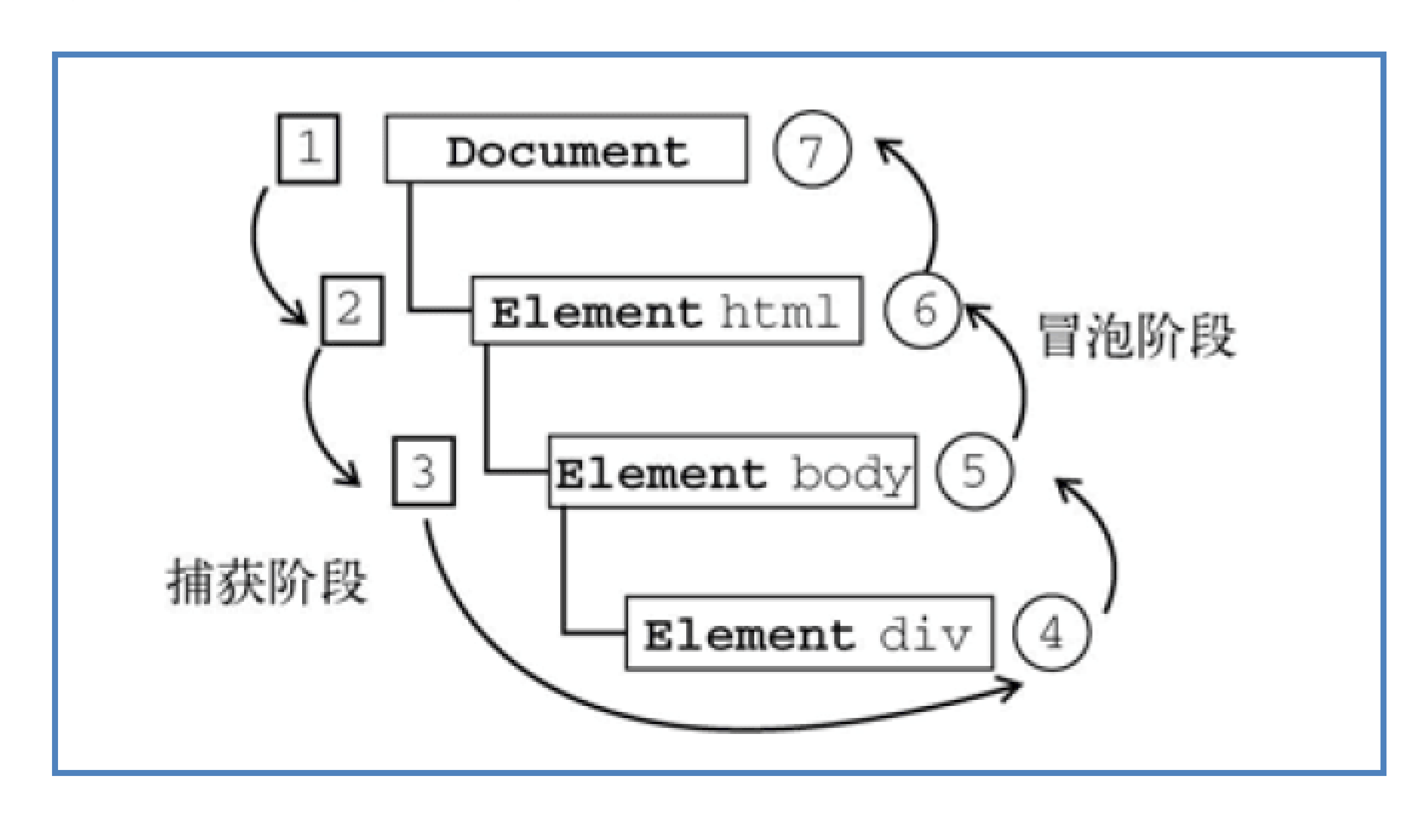


图 4-4　JavaScript 事件三阶段示意图



事件处理阶段，如果既注册了捕获事件，也注册了冒泡事件，这个时候事件的执行是按照事件注册的先后顺序来的。

4.2 剖析 React 事件系统

React 事件系统在原生的 DOM 事件体系上做了一些优化，封装了一个“合成事件”层，事件处理程序通过合成事件进行实例传递。在 React 的事件系统中，没有把所有事件绑定到对应的真实 DOM 上，而是使用委托机制实现了一个统一的事件监听器，把所有的事件绑定到了最外层 document 上，然后再将事件进行分发。这样极大地减少了内存开销，使运行性能得到极大提升。

在合成事件中，React 提供了三种绑定事件的方法：组件上绑定、在构造函数中绑定、箭头函数绑定。

4.2.1 组件上绑定事件

在组件上直接绑定事件和以前原生在 HTML 中的 DOM 元素绑定类似，这里通过示例 4-3 进行讲述。

【示例 4-3　组件上绑定事件】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>组件上绑定事件</title>
    <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script
src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class HelloWorld extends React.Component {
        constructor(props) {
            super(props);
        }
        showName(){
            alert("Hello React");
        }
        render () {
            return(
                <div>
                    <button onClick={this.showName}>单击事件</button>
                    //组件上绑定单击事件
                </div>
            )
        }
    }
    ReactDOM.render(<HelloWorld />, document.getElementById('root'));
</script>
</body>
</html>
```

对于 React 中组件绑定事件和 HTML 中的实际 DOM 绑定事件，有一点需要注意：在 HTML 中绑定，事件类型都为小写，并且函数要放到引号里面，例如：

```
<button onclick="showName()" >单击事件</button>
```

示例 4-3 在浏览器中的运行效果如图 4-5 所示。

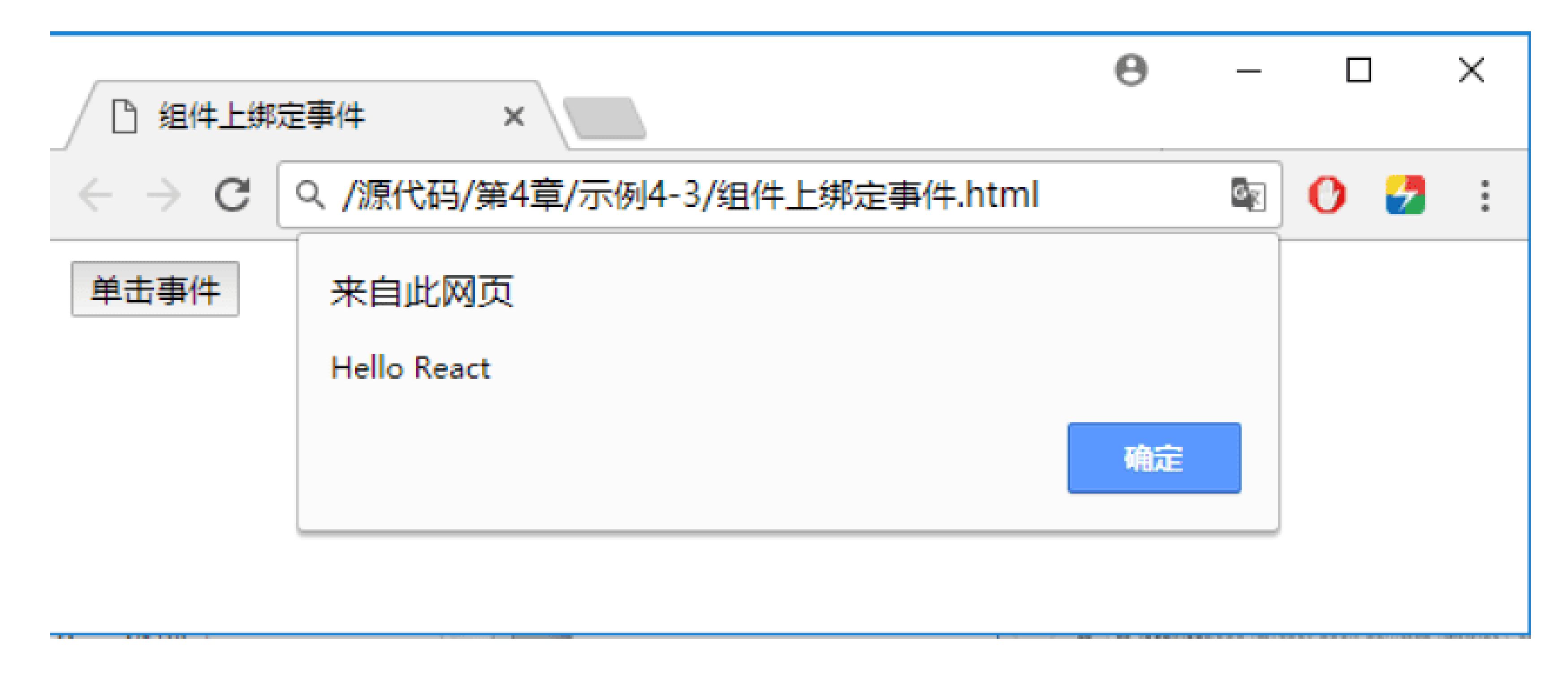


图 4-5　组件绑定单击事件

4.2.2　在构造函数中绑定事件

在构造函数中绑定方法，需要用 this 关键字来声明定义，如示例 4-4 所示。

【示例 4-4　构造函数中绑定事件】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>在构造函数中绑定事件</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class HelloWorld extends React.Component {
        constructor(props) {
            super(props);
            this.showName = this.showName.bind(this);
             //构造函数中 this 关键字声明事件
        }
        showName(){
            alert("Hello React");
        }
        render () {
            return(
                <div>
```

```
                    <button onClick={this.showName}>单击事件</button>
                </div>
            )
        }
    }
    ReactDOM.render(<HelloWorld />, document.getElementById('root'));
</script>
</body>
</html>
```

this 指向本组件，在构造方法中声明时，一定要用 bind()来绑定，传入 this 参数。如果没有绑定 this，在调用方法时会报错。Facebook 官方推荐使用该方式来绑定方法。示例 4-4 的运行结果和示例 4-3 的运行结果一致。

4.2.3　箭头函数绑定事件

ES6 新增的“箭头函数”可以用来绑定事件，如示例 4-5 所示。

【示例 4-5　箭头函数绑定事件】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>箭头函数绑定事件</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class HelloWorld extends React.Component {
        constructor(props) {
            super(props);
        }
        showName(){
            alert("Hello React");
        }
        render () {
            return(
```

```
                <div>
                    <button onClick={()=>this.showName()}>单击事件</button>
                      //箭头函数绑定事件
                </div>
            )
        }
    }
    ReactDOM.render(<HelloWorld />, document.getElementById('root'));
</script>
</body>
</html>
```

读者如果对箭头函数陌生，请查看本书第 1.6 节中有关 ES6 中的箭头函数介绍。示例 4-5 的运行效果和示例 4-3 的运行效果一致。

React 的合成事件基本涵盖了平时项目中所有的事件类型，比如剪贴板事件、键盘事件、鼠标事件、表单事件、滚轮事件等。详细内容可查看 Facebook 官方文档，链接地址为 https://reactjs.org/docs/events.html。

4.3 实战：实现登录界面（事件系统演练）

React 事件在实际项目中是必不可少的，本节笔者将用一个具体实例来讲解 React 的一些事件在具体应用中如何使用。React 的合成事件基本能够满足实际项目中的所有操作，在 4.2 节也提到了 React 官方文档中对事件有详细的阐述，读者可以去 React 官网查阅。

一些功能性网站都会有一个登录界面，输入正确的用户名和密码，可进入网站主页。这里就以登录界面为例，讲解 React 事件的使用。

本例登录界面的展示效果如图 4-6 所示。



图 4-6　登录界面效果

整体来看，要实现登录界面，需要定义一个组件来呈现需要展示的页面元素。组件需要定义自己的样式来修饰自身组件。另外就是事件绑定，比如用户名和密码不能超出 10 个字符，如果不符合要求，就提示用户重新输入；登录按钮需要绑定单击事件，提醒用户是否登录成功。这里先展示代码，如示例 4-6 所示。

【示例 4-6　登录界面事件讲解】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登录界面</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body style="background-color: #cccccc">
<div id="root"></div>
<script type="text/babel">
    class Login extends React.Component{
        constructor(props) {
            super(props);
            this.login = this.login.bind(this);
            this.check = this.check.bind(this);
            this.state={  //默认用户名和密码都为 admin
                userName:"admin",
                passWord:"admin"
            }
        }
        login(){
            if(this.refs.user.value===this.state.userName&&this.refs.pwd.
value===this.state.passWord)  //判断用户输入的用户名和密码是否为 admin
                alert("登录成功");
            else
                alert("登录失败");
        }
        check(){
            if(this.refs.user.value.length>10)  //检测用户名是否超过 10 个字符
                alert("超出 10 个字符，请重新输入");
        }
        render () {  //首先定义组件样式
            var loginStyle={
                width: 400,
```

```
                height: 250,
                background: "#FFF",
                margin:"200px auto",
                position: "relative"
            };
            var hStyle={
                position:"absolute",
                left:95,
                top:-40,
                padding: 0,
                margin: 50,
            };
            var pStyle={
                textAlign: "center"
            };
            var userStyle={
                width: 200,
                height: 30,
                border:"solid #ccc 1px",
                borderRadius: 3,
                paddingLeft: 32,
                marginTop: 50,

            };
            var pwdStyle={
                width: 200,
                height: 30,
                border:"solid #ccc 1px",
                borderRadius: 3,
                paddingLeft: 32,
                marginTop: 5,
            };
            var buttonStyle={
                width: 232,
                height: 30,
                background: "#E9E9E9",
                border:"solid #ccc 1px",
                borderRadius: 3,
                textAlign:"center"
            };
            return(
                <div style={loginStyle}>
                    <h1 style={hStyle}>登录界面</h1>
                    <div>
                        <p style={pStyle}><input type="text" style={userStyle}
placeholder="用户名" ref="user" onChange={this.check}/></p>
                        <p style={pStyle}><input type="password" style={pwdStyle}
```

```
placeholder="密码" ref="pwd"/></p>
                        <p style={pStyle}><button style={buttonStyle}
onClick={this.login}>登录</button></p>
                     </div>
                 </div>
             )
         }
     }
     ReactDOM.render(<Login />,document.getElementById("root"));
  </script>
  </body>
  </html>
```

首先在构造方法中声明 state，默认的用户名和密码为 admin，当用户输入的用户名和密码都为 admin 时，方可登录成功。另外，需要在构造方法中声明事件，这样的性能为最高效。绑定事件的 3 种方法在 4.2 节中已讲述，读者可返回去查看。React 中的组件样式，可定义成变量，可直接在组件中引用。在 React 中，可通过 ref 来获取 input 值。具体效果可在浏览器中查看。



在 JSX 中，样式中如果有“-”分割符，请写为小驼峰法，比如 text-align 需要写为 textAlign。

第 5 章

深入React原理

自从 React 在 2013 年 5 月开源到现在，其独特的设计思想、高性能的组织架构，一直吸引着千千万万的前端开发者。在实际的前端开发项目中，经常通过更新后台数据来重新渲染前端 UI。渲染 UI 就离不开 DOM 操作，然而经常性地操作 DOM 难免会导致项目性能变差。React 在这方面做了优化工作，提供了虚拟 DOM。要更新数据就先更新虚拟 DOM（虚拟 DOM 是内存数据，所以操作速度极快），再进行 dom-diff 操作，把最后真正发生变化的数据渲染到真实 DOM 中，整个过程大大减少了真实 DOM 操作，同时也大大提高了页面性能。本章将以 React 中几个重要的知识点来阐述 React 的工作原理。

5.1 JSX

虚拟 DOM 是 React 中的一个核心技术。在 JSX 之前，React 也提供了创建虚拟 DOM 的方法，例如要实现一个列表功能，可用 JavaScript 来实现，如示例 5-1 所示。

【示例 5-1　JavaScript 创建虚拟 DOM】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JavaScript 创建虚拟 DOM</title>
    <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    <script
src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class List extends React.Component{
        render(){
```

```
            var child1 = React.createElement('li', null, 'React');
            var child2 = React.createElement('li', null, 'Vue');
            var child3 = React.createElement('li', null, 'Angular');
            return(
               React.createElement('ul', { className: 'my-list' }, child1, 
child2, child3)
            )
         }
      }
      ReactDOM.render(<List/>,document.getElementById("root"));
   </script>
   </body>
   </html>
```

通过这种方式来创建虚拟 DOM，整体的可读性不是很好，如果 DOM 比较多，这种方式会显得异常凌乱。为了让代码的可读性更好，FaceBook 创造出一套完善的解决方案，那就是 JSX。

JSX 是 React 的重要组成部分，可以使用 XML 的方式来声明页面结构，是一种高效、优雅的语法糖。上述示例如果利用 JSX 来实现，可以像示例 5-2 这样写。

【示例 5-2　JSX 创建虚拟 DOM】

```
   <!DOCTYPE html>
   <html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>JSX 创建虚拟 DOM</title>
      <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
      <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
      <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
   </head>
   <body>
   <div id="root"></div>
   <script type="text/babel">
      class List extends React.Component{
         render(){
            return(
               <ul>
                  <li>React</li>
                  <li>Vue</li>
                  <li>Angular</li>
```

```
                </ul>
            )
        }
    }
    ReactDOM.render(<List/>,document.getElementById("root"));
</script>
</body>
</html>
```

虚拟 DOM 元素可以直接用 XML 方式来定义，这样比较符合以前的 HTML 中的 DOM 书写习惯，并且可读性较好。



用 JSX 书写代码时，一定要记住<script>标签中的 type 为"text/babel"，因为 JSX 是浏览器不识别的，需要利用 babel 来对 JSX 进行编译，转为浏览器能够理解的 JavaScript 代码。

5.1.1　JSX 语法

JSX 的语法和 XML 类似，可以定义自身属性以及子元素。另外，JSX 可以添加 JavaScript 表达式，用{}括起来。例如，在示例 5-2 中，可以把子元素定义为一个数组，直接用表达式来引用，代码如下：

```
<script type="text/babel">
    class List extends React.Component{
        render(){
            var lis = [
                <li key="li01">React</li>,  //<li>标签需要增加 key 属性，不然会报异常
                <li key="li02">Vue</li>,
                <li key="li03">Angular</li>
            ]
            return(
                <ul>
                    {lis}
                </ul>
            )
        }
    }
    ReactDOM.render(<List/>,document.getElementById("root"));
</script>
```

当然，JSX 中的{}还可以为一些计算的求值表达式，但是不能用 if-else 这样的条件判断语句，如果想实现条件判断功能，可以使用三目运算表达式，如示例 5-3 所示。

【示例 5-3　JSX 中的条件表达式】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JSX 条件表达式</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class Hello extends React.Component{
        render(){
            return(
                <div>Hello {this.props.name?this.props.name:"World"}</div>
                //三目运算，如果 name 属性不为空，则显示 name 属性值，否则显示 World
            )
        }
    }
    ReactDOM.render(<Hello name="React"/>,document.getElementById("root"));
</script>
</body>
</html>
```

5.1.2　JSX 使用样式

在实际项目中，有些组件的样式需要独立，那 CSS 样式在组件中该如何书写呢？按照以前传统的写法，可以把样式写到标签中的 style 属性中，比如：

```
<button  style="background-color: red">按钮</button>
```

在 JSX 中，样式需要用一个对象来保存，然后用{}进行引用，如示例 5-4 所示。

【示例 5-4　JSX 样式】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JSX 条件表达式</title>
```

```
        <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component{
            render() {
                var bgColor = {
                    backgroundColor:"red"
                }
                return(
                    <button style={bgColor}>按钮</button>
                )
            }
        }
        ReactDOM.render(<Hello name="React"/>,document.getElementById("root"));
    </script>
    </body>
    </html>
```

当然，也可以直接在标签中定义样式，代码如下：

```
render() {
    return(
        <button style={{backgroundColor:"red"}}>按钮</button>
    )
}
```

第一个{}为 JSX 语法，第二个{}表示对象。

5.2 dom-diff

从广义上讲，Web 界面的变化，实质上是数据的变更。数据可以为 DOM 节点、组件属性等，数据操作基本分为增加、删除、修改。在 React 中，UI 进行更新时，首先需要比对当前数据和前一状态的数据，哪些数据发生了变化，就把发生变化的数据渲染在 UI 上。

Web 界面实质上是构建的一棵 DOM 树，当某一节点发生变化时，React 会对当前的 DOM

树和前一状态的 DOM 树进行比较，这个比较的算法就是本节讲述的 dom-diff 算法。

其实 dom-diff 算法在 React 之前就已经有了，称为标准 dom-diff 算法。标准 dom-diff 算法是针对任意两棵树找最小变化步骤，这样的算法时间复杂度为 $O(n^3)$，目前市场上一些 Web 前端项目非常复杂，DOM 节点可能有成千上万个，显然这种方式的 diff 算法满足不了用户需要的性能。

React 在标准 dom-diff 算法上进行了优化，让时间复杂度从 $O(n^3)$减少到了 $O(n)$，这样 UI 的渲染性能就得到了极大提升。在理解 React 的 diff 算法时，读者要先了解一下 React 的 diff 算法是有两个假设的，官方的说法是：

1. Two elements of different types will produce different trees。

2. The developer can hint at which child elements may be stable across different renders with a key prop。

关于上述两条假设，在这里用通俗的语言再解释一下：

- React 认为相同类型的两个组件有类似的 DOM 树结构，在这种情况下会采用 diff 算法比较两个 DOM 树的差异。如果两个组件的类型不同，那么 React 会认为这两个 DOM 树结构不同，将之前的组件直接删除，然后创建新组件。
- 同一层次的一组节点，可以通过唯一的 key 值来进行区分。

基于上述两条假设，React 的 diff 算法就可以将算法复杂度减少到 $O(n)$。React 对 DOM 树进行了分层，只会对同一层的节点进行数据比较。另外，React 认为在 Web UI 中 DOM 节点的跨层级移动操作比较少，甚至可以忽略不计。React 的同层级比较如图 5-1 所示。

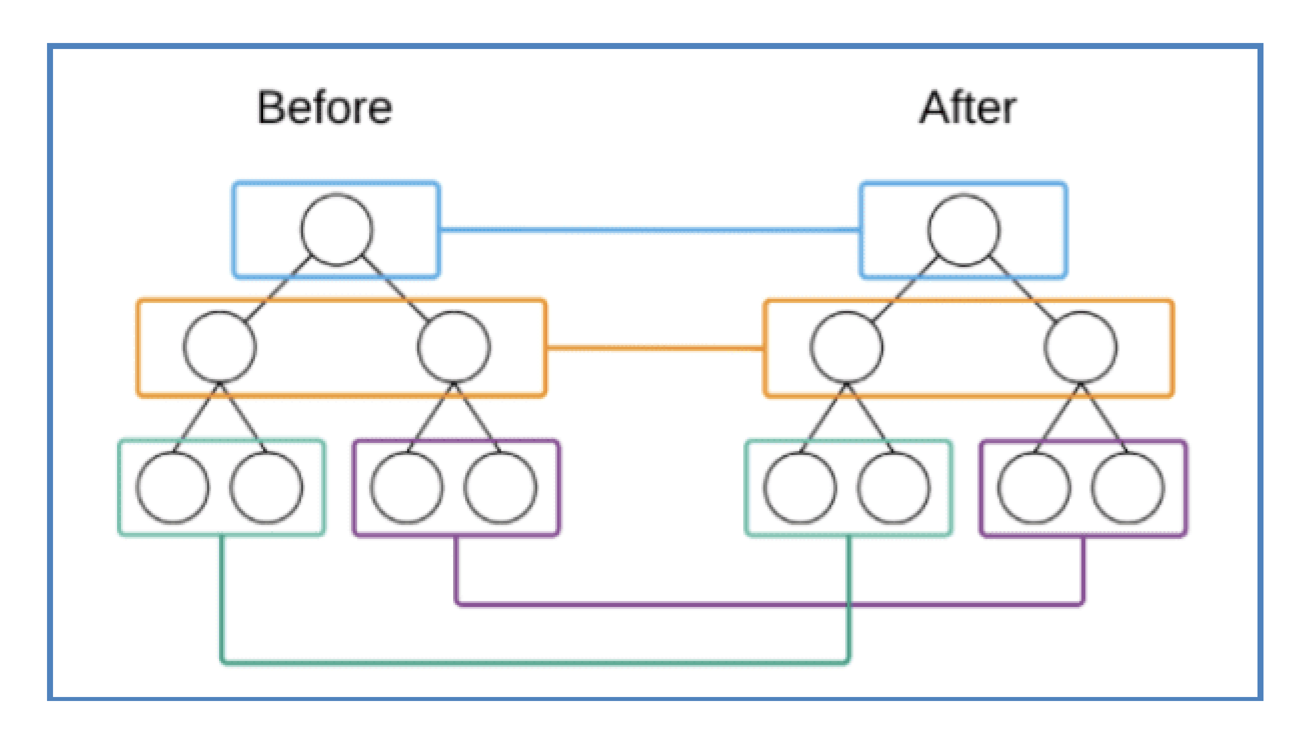


图 5-1　React DOM 树同层比较示意图

React 只会比较同一父节点下的子节点，即图 5-1 中相同颜色的节点。如果节点类型发生变化，那么 React 会将其删除，然后新建节点到新的 DOM 树上。如果节点类型相同、属性不同，那么 React 会进行替换操作。



读者在进行实际开发时，如果遇到同一层级的子节点进行操作时，需要加上 key 属性来进行唯一区别，否则 React 会进行告警。key 的唯一属性是避免删除、创建等重复操作，减少性能消耗。

这里用一个具体示例来解释一下 React 的 dom-diff 算法。假如现在有一棵 DOM 树，节点变化过程如图 5-2 所示。

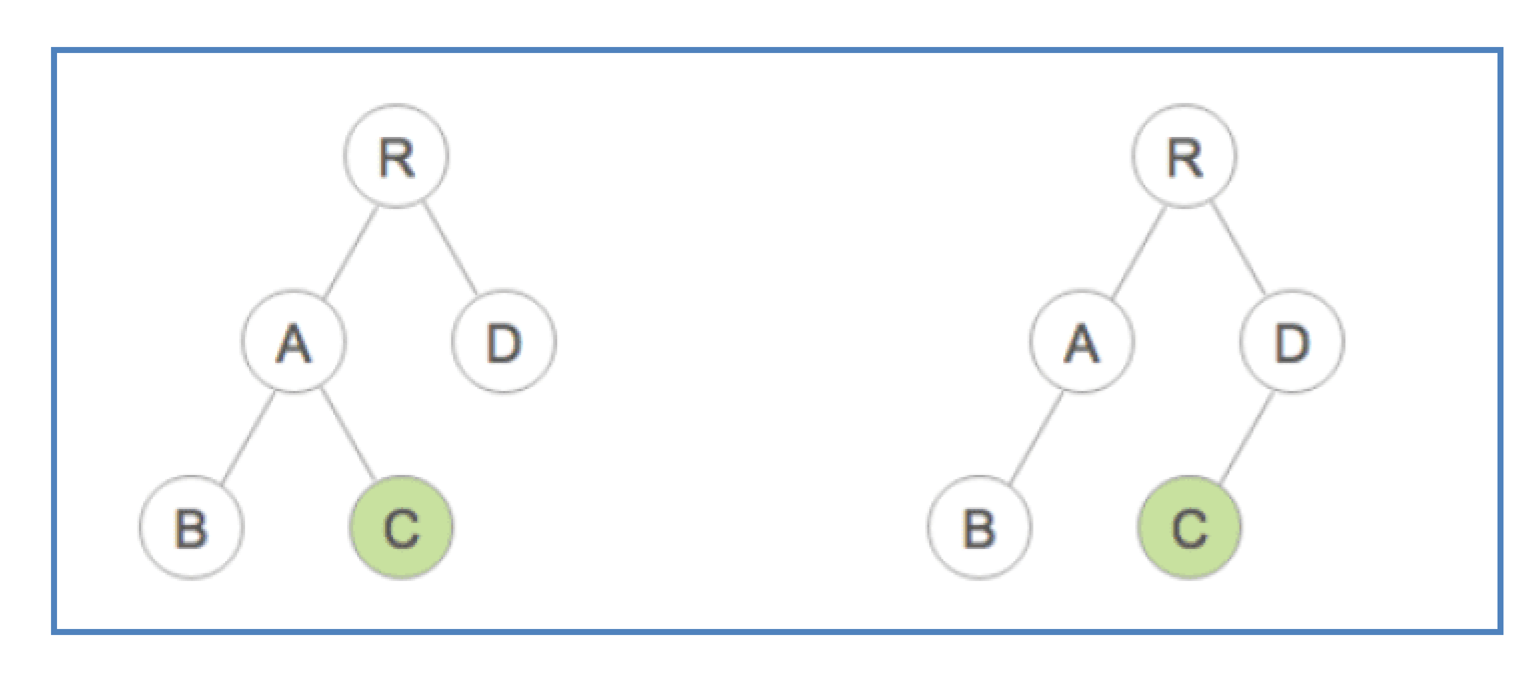


图 5-2　DOM 树节点变化过程

图 5-2 左侧为变化前的 DOM 树结构，右侧为变化后的 DOM 树结构，整个变化过程为：C 节点从 A 节点转移到了 D 节点上。React 对整棵 DOM 树的操作步骤为：

（1）删除 C 节点。
（2）创建 C 节点。
（3）更新 B 节点。
（4）更新 A 节点。
（5）渲染 C 节点。
（6）更新 D 节点。
（7）更新 R 节点。

5.3 setState

React 组件可以理解为一个状态机。组件的更新其实就是内部 state 值的更新，state 属性记录着组件的状态，在实际项目中会经常性地对组件进行重新渲染，这就离不开重新设置 state 属性。本节将讲述 React 修改 state 的方法 setState。

下面先通过一个示例来了解一下 setState 的用法。新浪微博有一个功能，可以给某一条微博进行点赞，也可以取消点赞，本示例简单模拟点赞功能。

【示例 5-5　setState 用法】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>setState 用法</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
```

```
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class Hello extends React.Component{
            constructor(){
                super();
                this.state={
                    isLiked:true
                }
                this.changeState = this.changeState.bind(this);
            }
            changeState(){
                this.setState({  //修改 state 值
                    isLiked :!this.state.isLiked
                })
            }
            render() {
                return(
                        <button onClick={this.changeState}>{this.state.isLiked?"点
赞":"取消"}</button>
                )
            }
        }
        ReactDOM.render(<Hello name="React"/>,document.getElementById("root"));
    </script>
    </body>
    </html>
```

组件状态默认值在构造函数中声明，通过单击按钮触发事件，改变 state 值。在上述示例中，state 中的 isLiked 属性默认值为 true，页面初始化时按钮上的内容为“点赞”，用户单击按钮，调用 changeState()方法中的 setState 来对状态值进行修改，此时 React 重新调用 render()方法对组件进行渲染。

setState()方法可以传入两个参数，格式如下：

```
setState(updater, [callback])
//updater 为新的 state 或 props，既可以为一个对象，也可以是一个函数，[callback]为回调函数
```

setState()方法为异步操作，实质上是通过一个队列机制来更新 state。在示例 5-5 中，在执

行 changeState()方法中的 this.setState 时，React 先把新 state 值放到了一个状态队列中，并没有及时更新 state 值。在示例 5-5 中加入一些 log 日志，可以看到这一现象，代码如下：

```
<script type="text/babel">
    class Hello extends React.Component{
        constructor(){
            super();
            this.state={
                isLiked:true
            }
            this.changeState = this.changeState.bind(this);
        }
        changeState(){
            console.log(this.state.isLiked)  //输出当前 state 值
            this.setState({
                isLiked :!this.state.isLiked
            })
            console.log(this.state.isLiked)  //setState 后，输出当前 state 值
        }
        render() {
            return(
                    <button onClick={this.changeState}>{this.state.isLiked?"点
赞":"取消"}</button>
            )
        }
    }
    ReactDOM.render(<Hello name="React"/>,document.getElementById("root"));
</script>
```

在 changeState()方法中添加两行 log 日志，分别放在 setState()方法前后。这里先假设 setState()为同步更新，在构造方法中初始化 state 时，isLiked 值为 true，页面刷新后，按钮上的默认值为“点赞”。当单击按钮调用 changeState()方法时，第一个 log 日志输出的 this.state.isLiked 值为 true，这个毫无疑问，然后执行 this.setState()方法，对 state 属性进行更新操作，再输出第二个 log 日志，如果 setState()方法为同步更新，那么第二个 log 日志输出应该为 false，在这里看一下运行结果，如图 5-3 所示。

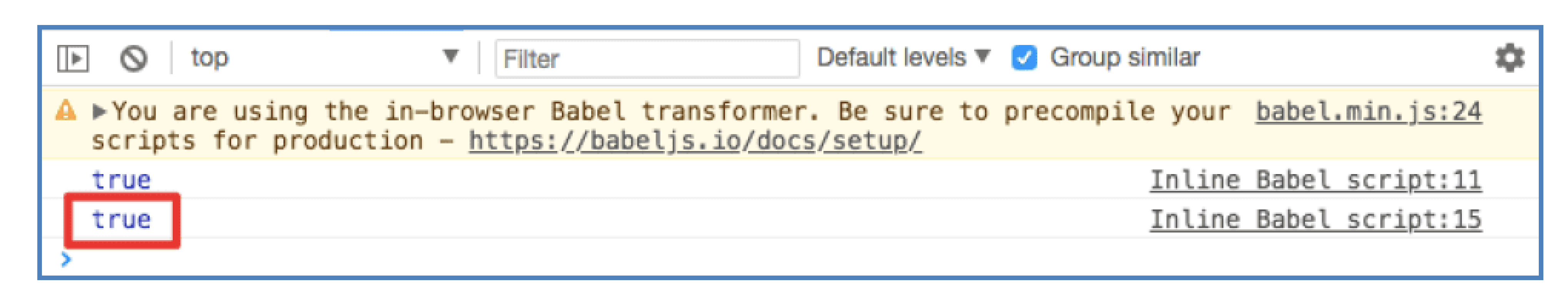


图 5-3　setState()方法异步测试

可以看到第二个 log 日志输出的 this.state.isLiked 值为 true，仍然是旧状态值。这种反证法

可以说明 setState()方法确实为异步操作。

React 中的 setState 异步原理是什么呢？笔者这里再举一个示例（见示例 5-6），对 setState 的操作原理进行进一步讲解。

【示例 5-6　setState 原理】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>setState 原理</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    class HelloWorld extends React.Component {
        constructor(props) {
            super(props);
            this.state = {
                count:0
            };
            this.showState = this.showState.bind(this);
        }
        componentDidMount() {
            this.setState({
                count:this.state.count+1
            });
            this.setState({
                count:this.state.count+1
            });
            this.setState({
                count:this.state.count+1
            });
            console.log("setState 后立即显示 state 值为："+this.state.count);
        }
        showState(){
            console.log("showState 方法中的 state 值为："+this.state.count);
        }
```

```
        render () {
            return(
                <button onClick={this.showState}>显示当前 state 值</button>
            )
        }
    }
    ReactDOM.render(<HelloWorld />, document.getElementById('root'));
</script>
</body>
</html>
```

上述示例，在组件挂载完成后，进行三次 setState.count+1 操作，立即显示当前 state.count 值，再通过单击事件来显示当前 state.count 值，程序运行效果如图 5-4 所示。

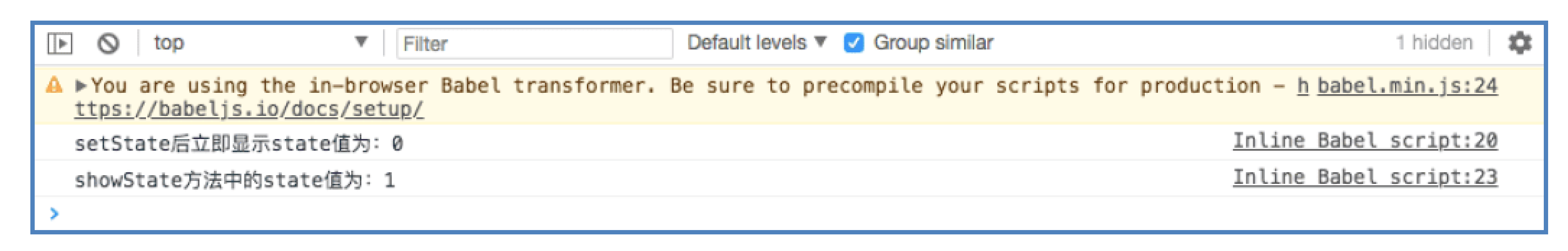


图 5-4　state 异步执行效果

从图 5-4 显示的结果可以看出，虽然在 componentDidMount()方法中执行了三次 setState，但是最后就执行了一次，并且没有立即执行。原因是 React 先把新的 state 值放到了状态队列里，最后对三次 setState 进行了批量处理，批处理过程中进行了合并，所以结果就是对 setState 只执行了一次，这样做的好处是可以避免一些重复渲染，这也是 React 能够提升性能的一个原因。对于 setState，React 官方是这么说的：

Think of setState() as a request rather than an immediate command to update the component. For better perceived performance, React may delay it, and then update several components in a single pass. React does not guarantee that the state changes are applied immediately.

大概可以理解为：setState()可以理解为一个重新渲染的请求，而不是立即更新的一个命令，为了有更好的性能，React 可能会推迟执行，然后会批量进行处理。React 不会保证在 setState 操作后立即拿到最新的 state 值。

如果想在 setState 后立即得到最新的 state 值，可以通过函数来实现，如示例 5-7 所示。

【示例 5-7　setState 传入函数】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>setState 传入函数</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
```

```
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class HelloWorld extends React.Component {
            constructor(props) {
                super(props);
                this.state = {
                    count:0
                };
                this.showState = this.showState.bind(this);
            }
            componentDidMount() {
                this.setState(
                    function(state) {
                        console.log("第一次 setState 后的 count 值: "+state.count);
                        return {
                            count: state.count + 1
                        }
                    }
                );
                this.setState(
                    function(state) {
                        console.log("第二次 setState 后的 count 值: "+state.count);
                        return {
                            count: state.count + 1
                        }
                    }
                );
                this.setState(
                    function(state) {
                        console.log("第三次 setState 后的 count 值: "+state.count);
                        return {
                            count: state.count + 1
                        }
                    }
                );
            }
            showState(){
```

```
            console.log("showState 方法中的 state 值为: "+this.state.count);
        }
        render () {
            return(
                <button onClick={this.showState}>显示当前 state 值</button>
            )
        }
    }
    ReactDOM.render(<HelloWorld />, document.getElementById('root'));
</script>
</body>
</html>
```

如果 setState 的第一参数为函数时，那么该函数会接收到组件前一时刻的 state 值，并将其作为入参，计算返回的是最新的 state 值。setState 第一个参数为函数时，React 的处理方式和入参为对象的处理方式不同，React 会将函数放到一个任务队列中依次执行，所以每个函数执行后返回的 state 值都是当前最新的。示例 5-7 的运行结果如图 5-5 所示。

```
▶You are using the in-browser Babel transformer. Be sure to precompile your scripts for production - https://babeljs.io/docs/setup/   babel.min.js:24
第一次setState后的count值：0   Inline Babel script:13
第二次setState后的count值：1   Inline Babel script:21
第三次setState后的count值：2   Inline Babel script:29
showState方法中的state值为：3   Inline Babel script:37
```

图 5-5　setState 传入函数 state 可同步



读者在实际项目中不要用 this.state 来修改 state 值，因为获取到的 this.state 有可能不是最新的，更新组件 state 时要用 setState()方法。

第 6 章

◀React组件编写实战▶

组件是 React 这个前端框架的一个灵魂，比如盖一座房子需要门、窗户、房梁等，每一个小的部分都是构成房子的重要因素。组件就是 React 的每一个小组成，一个优秀的 React 项目必然是由很多优秀的组件来构建的。本章将带领读者进入 React 世界里的组件部分。

6.1 React 组件写法

在更好地使用 React 组件之前，读者应该掌握 React 组件的写法。React 组件写法有三种方式（React createClass 写法、React、Component 写法和无状态函数写法），选择一种更适合的编码方式，在一定程度上会使得项目开发更加高效。比如从一个出发点到一个目的地，两点之间有很多条路，选择一条更短的路，会减少路途上的行走时间。

6.1.1 React.createClass 写法

这种写法是 ES5 写法，中规中矩，比较符合以前的 JavaScript 定义类的编码习惯，这种组件写法也是最早 React 官方推荐的定义组件方法（后来 ES6 的盛行，目前最推崇的方法是 ES6 的组件写法，后面讲述）。这里以 HelloWorld 组件的简单操作来介绍这种组件写法，如示例 6-1 所示。

【示例 6-1　组件写法-React.createClass】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>组件写法--React.createClass</title>
    <script src="https://cdn.bootcss.com/react/15.4.2/react.min.js">
</script>  //这里用的 React 的 15 版本
    <script src="https://cdn.bootcss.com/react/15.4.2/react-dom.min.js">
</script>
    <script src="https://cdn.bootcss.com/babel-standalone/6.22.1/
babel.min.js"></script>
</head>
```

```
<body>
<div id="root"></div>
<script type="text/babel">
    var HelloWorld = React.createClass({
        getInitialState: function(){  //获取 state 的初始值
            return{
                message:"单击显示信息"
            };
        },
        componentWillMount: function(){
            console.log("1. 挂载前执行")
        },
        componentDidMount: function(){
            console.log("2. 挂载后执行")
        },
        showMessage:function(){
            alert("Hello React");
        },
        render:function() {
            return (
                <button onClick={this.showMessage}>{this.state.message}</button>
            )
        }
  });
  ReactDOM.render(<HelloWorld />,document.getElementById("root"));
</script>
</body>
</html>
```

注意，在该示例中，笔者引用的 React 版本为 15，本书主要讲解的 React 版本为 16。那为什么这个示例中用的 React 版本会是以前的版本呢？原因是最新的 React 版本 16 已经不支持这种 React 组件写法了，所以读者对这种组件写法了解一下即可，因为目前网上一些教程还是以这种写法为例，能够理解就行。目前组件的主流写法是下面讲述的 ES6 的写法。

6.1.2 React.Component 写法

这种是 ES6 写法，是目前 React 官方主推写法，并且 React 16 版本已经废弃了 ES5 的 React.createClass 写法。如果读者有过 Java 编程经验，那这种写法学起来会更加轻松，因为 ES6 引入 class 之后，基本和 Java 的写法类似。这里还是以 HelloWorld 组件为例讲述该写法，如示例 6-2 所示。

【示例 6-2　组件写法-React.Component】

```
<!DOCTYPE html>
```

```
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>组件写法--React.Component</title>
        <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class HelloWorld extends React.Component{
            constructor(){
                super()
                this.state = {
                    message:"单击显示信息"
                }
            }
            componentWillMount(){
                console.log("1. 挂载前执行")
            }
            componentDidMount(){
                console.log("2. 挂载后执行")
            }
            showMessage(){
                alert("Hello React");
            }
            render(){
                return (
                    <button onClick={this.showMessage}>{this.state.message}</button>
                )
            }
      }
      ReactDOM.render(<HelloWorld />,document.getElementById("root"));
    </script>
    </body>
    </html>
```

ES6 的写法，整体看起来优雅大气，并且很符合类的写法。另外，React.Component 的组件定义在性能上也优于 React.createClass 的组件定义，随着 React 的发展，ES6 的写法会是一

个主流，建议读者在平时项目开发中使用 ES6 的组件写法。

6.1.3 无状态函数写法

所谓无状态组件，就是没有 state 属性的参与，数据只接收 props 属性值。为什么要使用无状态的组件定义呢？随着 Web 页面的需求不断增加，某些项目的功能可能极其复杂，为了代码减少耦合度，并且让每个组件各尽其职，需要某类组件只负责呈现数据，不需要更多的逻辑操作，这个时候无状态组件就诞生了。具体用法如示例 6-3 所示。

【示例 6-3 组件写法-无状态函数】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>组件写法--无状态函数</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/babel.min.js">
</script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
  const HelloWorld = (props)=>{
      return(
          <div>
              hello {props.name}
          </div>
      )
  }
  ReactDOM.render(<HelloWorld name="react"/>,
document.getElementById("root"));
</script>
</body>
</html>
```

箭头函数是 ES6 的新标准，已在 1.6 节中讲述，如果读者对箭头函数陌生，可以返回查看。除了传递 props，还可以增加参数，例如可以这么写：

```
const HelloWorld = ({onClick,text,...props})=>{
     return(
```

```
            <div>
                hello {props.name}
            </div>
        )
}
```

6.2 React 组件分类

在一定程度上，React 的组件化对传统的大型 Web 项目进行了架构优化，各个组件实现特定功能，并且组件之间耦合度非常低。为了能够使组件进一步细化功能，就有了 React 组件分类的概念。React 的组件分类，主要是以两方面为标准的：一方面是呈现，一方面是管理。这就是本节中将要讲述的两类组件：木偶组件（Dumb Component）和智能组件（Smart Component）。

6.2.1 木偶组件和智能组件

木偶的含义为呆板，能够理解的事物很少。React 的木偶组件的含义类似，负责功能比较单一，主要是担任呈现 UI 职责。智能的含义为灵活，能够处理各种各样的复杂操作。React 的智能组件用于处理一些复杂的逻辑操作。为什么要进行这样的分类呢？

在项目开发中，经常会遇到这种情况：通过后台接口从数据库获取个人信息，然后将个人信息数据陈列出来。这里就以该场景为基础，实现一个列表的数据展示的功能，如示例 6-4 所示。

【示例 6-4　列表展示数据】

将要展示的数据先保存到一个 json 文件中，名为 data.json，内容如下：

```
[
  {"name":"React","id":"01"},
  {"name":"Vue","id":"02"},
  {"name":"Angular","id":"03"}
]
```

每一项都有两个属性，分别是 name 属性和 id 属性：name 是用来展示呈现的，id 是用来区分同级同组节点的。下面是项目代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>列表展示数据</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
```

```
react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
        <script src="jquery-3.3.1.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        class CommentList extends React.Component {
            constructor(props) {
                super(props);
                this.state = { languages: [] }

            }
            componentDidMount() {
                $.ajax({  //通过 ajax 请求获取 json 文件中的数据
                    url: "data.json",
                    dataType: 'json',
                    success: function(languages) {
                        this.setState({languages: languages});
                    }.bind(this)
                });
            }
            render() {
                return(
                    <ul>
                        {this.state.languages.map(function(language) {
                          //通过 map 方法循环列表
                            return (
                                <ul key={language.id}>
                                    <li >{language.name}</li>
                                </ul>
                            );
                        })}
                    </ul>
                )
            }
        }
      ReactDOM.render(<CommentList />,document.getElementById("root"));
    </script>
    </body>
```

```
</html>
```

上述示例可以分为两部分：一部分为 ajax 请求获取 json 文件内容，另一部分是将获取到的数据进行 UI 渲染。读者有没有觉得这种组件的实现方式有些不妥？如果数据请求来源很多并且处理数据步骤很多，将所有代码都放在一个组件中，是不是很“臃肿”呢？另外，React 组件的设计思想就是单一化功能，把 ajax 获取数据的部分和 UI 渲染放在一起，是不是有点不妥呢？如果把获取数据和 UI 渲染分开，分别放在一个独立的组件中，代码的可读性、可维护性、可复用性是不是更好呢？接下来我们将两部分分开，如示例 6-5 所示。

【示例 6-5　木偶组件和智能组件】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>木偶组件和智能组件</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    <script src="jquery-3.3.1.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    //木偶组件
    class CommentList extends React.Component{
        constructor(props){
            super(props);
        }
        render() {
            console.log(this.props)
            return(
                <ul>
                    {this.props.languages.map(function(language) {
                        return (
                            <ul key={language.id}>
                                <li >{language.name}</li>
                            </ul>
                        );
                    })}
                </ul>
```

```
            )
        }
    }
    //智能组件
    class CommentContainer extends React.Component{
        constructor(){
            super();
            this.state = {
                languages:[]
            };
        }
        componentDidMount(){
            $.ajax({
                url: "data.json",
                dataType: 'json',
                success: function(languages) {
                    this.setState({languages: languages});
                }.bind(this)
            });
        }
        render(){
            return <CommentList languages={this.state.languages}/>
        }
    }
    ReactDOM.render(<CommentContainer />,document.getElementById("root"));
</script>
</body>
</html>
```

按照不同的关注点将内容分离，木偶组件只关注接收数据并将其呈现，不涉及数据操作，而智能组件需要考虑更多的逻辑操作性问题。这样有利于组件复用和维护，前端的项目，界面 UI 会经常性发生变动，但是其具体操作是业务逻辑，变化很少，而木偶组件主要负责 UI 呈现，一旦 UI 发生变化，只修改木偶组件即可。示例 6-5 的运行效果如图 6-1 所示。



图 6-1　木偶组件和智能组件

木偶组件和智能组件也称作呈现组件（Presentational Component）和容器组件（Container Component）。读者如果在网上看到这样的概念，要知道是一回事。

- 木偶组件的特点：只关注 UI 呈现，不关心数据操作及来源，更像是一个 UI 接口。

通常没有自己的 state 属性，数据主要是通过 props 来获取。木偶组件通常可以写为无状态的函数组件（可参考 6.1 节）。

- 智能组件的特点：只关注事务逻辑操作，有自己的 state，并且不关注 UI 怎么呈现。

6.2.2　高阶组件

在介绍高阶组件之前，先带领读者理解一下高阶函数。所谓高阶函数，就是可以以一个函数为入参，返回结果也可能是函数的一个复杂函数。其实之前用到的 setTimeout()、Array.map()都是高阶函数。这里自定义一个简单高阶函数（见示例 6-6），可能会更好理解。

【示例 6-6　高阶函数】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>高阶函数</title>
    <script type="text/javascript">
        function add(x,y) {
            return x+y;
        }
        function higherFunc(x,y,f) {
            return f(x,y)
        }
        console.log(higherFunc(1,2,add))
    </script>
</head>
<body>
</body>
</html>
```

对上述示例进行一下分析：add 为一个自定义函数，实现两数相加功能，并返回结果。higherFunc 也是一个自定义函数，但是有三个参数，并且第三个参数为函数。在执行 higherFunc 函数时，可以推导运行过程：

```
x = 1
y = 2
f = add(1,2) =>结果为 3
return f(1,2) =>结果为 3
```

所以上述示例中，log 输出结果为 3。

高阶组件的官方定义如下：

A higher-order component is a function that takes a component and returns a new component。

通俗地讲，高阶组件就是一个函数，其参数可以接收一个组件，然后可以返回一个组件，用以下方式表示：

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

高阶组件可以理解为一个工厂模式，现有一些共同的基础组件，然后经过工厂的加工，可以得到一个新的组件。WrappedComponent 就是一些共同的基础组件，higherOrderComponent 为一个工厂，EnhancedComponent 就是加工后的新组件。下面通过示例 6-7 来了解一下高阶组件的用法。

【示例 6-7　高阶组件的应用】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>高阶组件的应用</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    function HOC(WrappedComponent) {  //高阶组件，用来加工被包装组件
        return class extends React.Component{
            render(){
                return (
                    <div>
                        <h1>这是标题</h1>
                        <WrappedComponent />
                    </div>
                )
            }
        }
    }
    class HelloWorld extends React.Component{  //要被包装的组件
        render(){
            return <div>这是内容</div>
        }
    }
```

```
    var NewComponent = HOC(HelloWorld);
    ReactDOM.render(<NewComponent/>,document.getElementById("root"));
</script>
</body>
</html>
```

在上述示例中，<HelloWorld/>是需要被包装的组件，其渲染只有“<div>这是内容</div>”，效果如图 6-2 所示。如果需要在该组件上进行进一步的加工，就可以利用高阶组件来对其进行处理。HOC 就是一个可以对基础组件进行加工的高阶组件。



图 6-2　高阶组件为基础组件添加标题

对于上述示例，读者还可以打开浏览器的开发者工具，对这一结构进行分析，如图 6-3 所示。

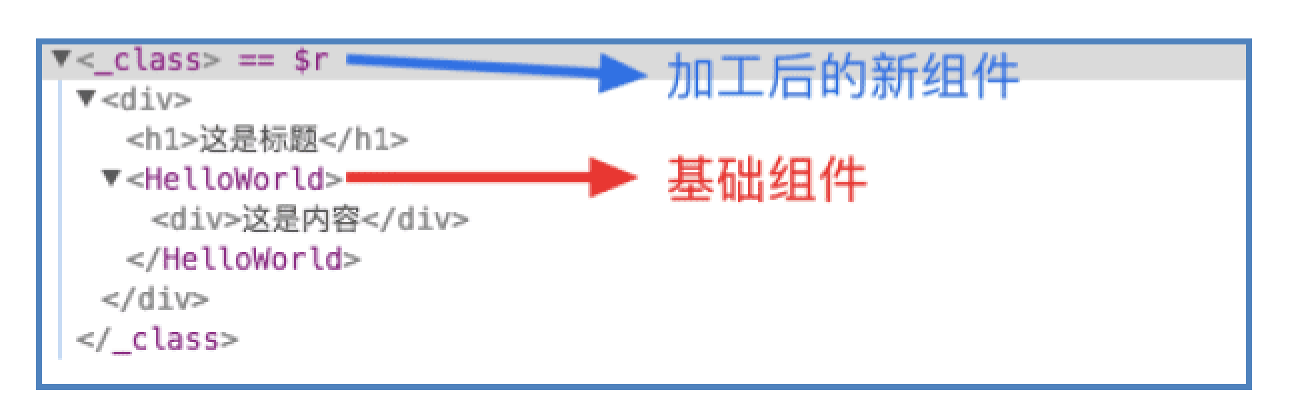


图 6-3　高阶组件 DOM 结构

关于高阶组件的具体用法，目前流传的有两类用法：一类是属性代理（Props Proxy，PP），另一类是反向继承（Inheritance Inversion，II）。

（1）属性代理

基础组件，也就是需要被包装的组件，其自身可能会有一些 props 和 state 属性，进入高阶组件这个工厂后，props 有可能会被工厂私自修改，加工后得到的新组件 props 或者 state 是修改后的属性值。这就是属性代理的含义。这里以示例 6-8 来进行分析。

【示例 6-8　高阶组件的属性代理】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>高阶组件的属性代理</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
```

```
        <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
    </head>
    <body>
    <div id="root"></div>
    <script type="text/babel">
        function HOC(WrappedComponent) {
            return class extends React.Component{
                render(){
                    const newProps={  //定义新 props
                        name:"Hello React!",
                        language:"JavaScript"
                    }
                    return (
                        <div>
                            <h1>这是标题</h1>
                            <WrappedComponent {...this.props} {...newProps}/>
                        </div>
                    )
                }
            }
        }
        class HelloWorld extends React.Component{
            static defaultProps={
                name : "Hello World!"
            }
            componentDidMount(){  //输出组件属性
                console.log(this.props);
            }
            render(){
                return <div>{this.props.name}</div>
            }
        }
        var NewComponent = HOC(HelloWorld);
        ReactDOM.render(<NewComponent/>,document.getElementById("root"));
    </script>
    </body>
    </html>
```

原始组件的 props.name 值为“Hello World！”，经过属性代理后，运行结果如图 6-4 所示。

```
⚠ ▸You are using the in-browser Babel transformer. Be sure to precompile your scripts for production – https://babeljs.io/docs/setup/    babel.min.js:24
  ▸{name: "Hello React!", language: "JavaScript"}                                                       Inline Babel script:23
>
```

图 6-4　属性代理后原始组件 props 发生改变

可以看到，原始组件的 props 发生了变化，原始的 props.name 变为“Hello React！”，并且增加了一个属性 props.language。

（2）反向继承

高阶组件中传入的基础组件会被高阶组件继承。这样的话，基础组件成为父类，高阶组件成为子类，这就是反向继承的意思。下面用示例 6-9 来进行该用法的介绍。

【示例 6-9　高阶组件的反向继承】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>高阶组件的反向继承</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/
react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script>
</head>
<body>
<div id="root"></div>
<script type="text/babel">
    function HOC(WrappedComponent) {
        return class extends WrappedComponent{
            componentDidMount(){
                this.setState({
                    isShow:false
                })
            }
            render(){
                return (
                    super.render()
                )
            }
        }
    }
    class HelloWorld extends React.Component{
        constructor(){
            super();
            this.state={
                isShow : true
            }
```

```
        }
        render(){
            console.log(this.state)  //查看 state 的变化过程
            return <div>{this.state.isShow?"Hello World":""}</div>
            //判断 state.isShow 值，如果为 true 则显示内容，反之不显示
        }
    }
    var NewComponent = HOC(HelloWorld);
    ReactDOM.render(<NewComponent/>,document.getElementById("root"));
</script>
</body>
</html>
```

HOC 高阶组件继承基础组件 HelloWorld，原始基础组件 HelloWorld 的 state.isShow 值为 true，经过 HOC 组件后将其 state 值进行了改变，最后的结果是将渲染的元素进行隐藏。这里需要注意，基础组件中的 render()方法中有 console.log 输出，结果如图 6-5 所示。

```
⚠ ▶You are using the in-browser Babel transformer. Be sure to precompile your scripts for production - https://babeljs.io/docs/setup/    babel.min.js:24
  ▶{isShow: true}                                                                                                    Inline Babel script:24
  ▶{isShow: false}                                                                                                   Inline Babel script:24
```

图 6-5　基础组件的 console.log 日志输出

读者有没有这样的疑问，为什么会输出两次，并且两次的输出结果不一样？这里需要注意，在高阶组件中修改 state 属性是在 componentDidMount()方法中执行的，第一次的渲染 state 属性并没有改变，等渲染后调用 componentDidMount()方法，这时执行 setState，React 会在执行 setState 后自动调用 render()方法，此时 state 中的属性 isShow 值已经改变成了 false，所以第二次的输出为 false。

第 7 章

◂ Redux数据管理 ▸

随着单页面（SPA）的发展以及 Web 前端项目的需求越来越复杂，数据管理成了一个非常困难的问题。对于一些大型项目，如果单纯依靠 React 来实现，后续的各种页面组件状态会变得越来越难管理，更何况 React 只是在 View 层的一个框架。其实对于数据管理，Facebook 在 2014 年提出了 Flux 架构的概念，也得到了很多前端开发者的追随，后续还有一些开发者对其进行了改进优化。到了 2015 年，Redux 诞生，将 Flux 进行了融合，很快成为前端很流行的开发框架。本章主要讲述 Redux 这一前端框架。

7.1 总览 React 数据管理

早期的 Web 项目需求比较简单，涉及的 UI 渲染主要来自 CSS 的修饰，界面数据相对简单一点。后来随着 Web 前端技术的发展，以及日益增长的用户需求，促发了许多新型框架的生产。其实读者可以想一想，很多新技术的出现一定是有一个前提的，那就是某个领域的发展是需要这个东西的，并且新技术能够火起来，一定是符合当时需求的。正如现在的 JavaScript 单页面应用，市场流行这样的项目结构，然而以前的框架满足不了这种开发需求。其实这种需求大多数来源于数据，因为现在的一些大型项目，数据非常庞大，如果没有一个合理的数据管理机制，那么项目会做的越来越难，后期维护更是无从下手。本节主要讲解关于状态管理的几种前端框架。

7.1.1 Flux 的出现

Flux 的出现是 Web 前端发展的一个必经过程，如果当时不是 Flux，那也会有其他的模式或者框架来填充这一块空白，因为当时的市场需求单从 React 层面已经无法满足。打个比方，一家小型互联网公司，人员组织架构比较简单，做的业务可能也相对不复杂，这个时候为了节约成本，可能一个人负责好几个人的工作职责，一个程序员有可能前端和后端一起做，但是如果公司组织结构比较庞大，并且市场业务比较多，如果还按照小公司的规章制度来管理，难免会出各种各样的问题。

现在读者应该知道了吧，React 的优势在于能够把 View 层分解成各个小部分，让整个项目更加模块化。传统的 MVC 模式，除了 View 层的实现，还有数据层和控制层的实现。当时

Facebook 也意识到，虽然 React 在 View 层做的已经很优秀了，但是在真正的项目中难免还会有 Model 层和 Controller 层的涉及，所以 2014 年 Facebook 推出了 Flux。其实 Flux 的出现，更像是在填充 React 的空白，协助 React 成为一个新的 MVC 模式。下面将带领大家了解 Flux 的组成部分以及 Flux 的具体应用。

Flux 主要涉及 4 个重要概念：

- Dispatcher：处理动作的一个分发器，是 Flux 应用程序中数据流的中心枢纽，主要任务是将收到的行为分发给 Store。
- Store：对数据进行管理。
- View：React 组件，主要负责 View 层。
- Action：提供给 Dispatcher，传递数据给 Store。

官方给出的 Flux 的整个结构如图 7-1 所示。

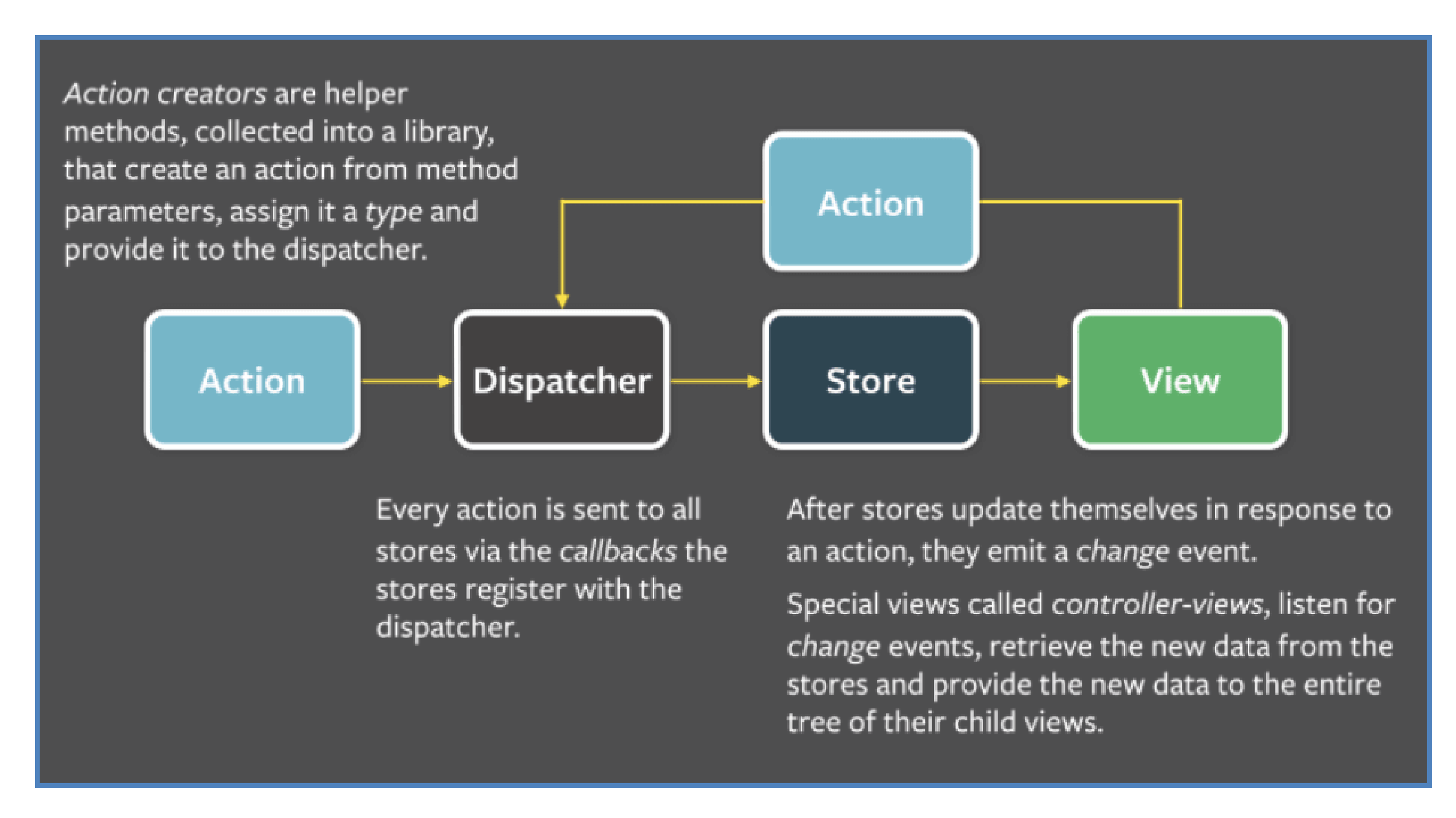


图 7-1　Flux 整体结构图

从结构图可以看到，在 Flux 的应用程序中，数据流是单向的，这样可以更好地控制数据状态。首先会产生一个事件，一般是由用户对界面执行的一个操作，Action 得到这个操作后，将其交给 Dispatcher，再由 Dispatcher 来进行分发给 Store，Store 收到通知后对相关数据进行维护，再发出一个更改通知，告诉视图层需要更新 View 了，然后重新从 Store 中检索数据，调用 setState 方法对 View 进行更新。这里以一个 Flux 官方示例对其进行讲解，如示例 7-1 所示。

【示例 7-1　Flux 示例】

该项目的核心目录结构如图 7-2 所示，这是 src 目录。

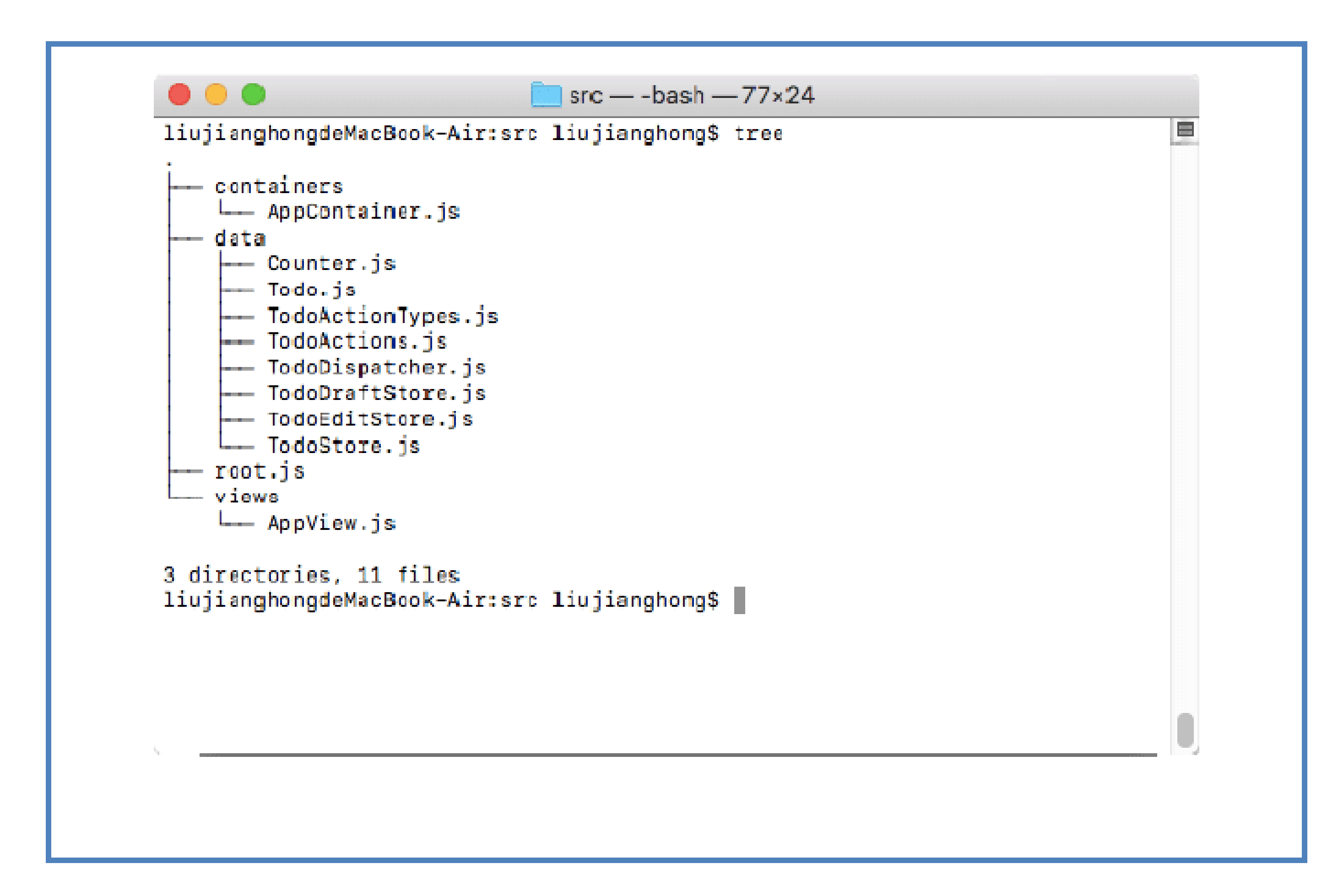


图 7-2　Flux 核心目录结构

（1）和 src 目录并列的还有 index.html，用来引用相关的样式文件和 JavaScript 文件，并定义需要显示的 DOM 元素。

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Flux • TodoMVC</title>
    <link rel="stylesheet" href="todomvc-common/base.css">
  </head>
  <body>
    <section id="todoapp"></section>
    <footer id="info">
      <p>Double-click to edit a todo</p>
      <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
    </footer>
    <script src="./bundle.js"></script>
  </body>
</html>
```

（2）依据 Flux 数据的流动过程，让用户对界面触发一个事件后，Action 会得到这一消息，下面看一下担任 Action 角色的 TodoAction.js 文件：

```
'use strict';
import TodoActionTypes from './TodoActionTypes';
import TodoDispatcher from './TodoDispatcher';
const Actions = {
  addTodo(text) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.ADD_TODO,
```

```
      text,
    });
  },
  deleteCompletedTodos() {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.DELETE_COMPLETED_TODOS,
    });
  },
  deleteTodo(id) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.DELETE_TODO,
      id,
    });
  },
  editTodo(id, text) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.EDIT_TODO,
      id,
      text,
    });
  },
  startEditingTodo(id) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.START_EDITING_TODO,
      id,
    });
  },
  stopEditingTodo() {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.STOP_EDITING_TODO,
    });
  },
  toggleAllTodos() {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.TOGGLE_ALL_TODOS,
    });
  },
  toggleTodo(id) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.TOGGLE_TODO,
      id,
    });
  },
```

```
  updateDraft(text) {
    TodoDispatcher.dispatch({
      type: TodoActionTypes.UPDATE_DRAFT,
      text,
    });
  },
};
export default Actions;
```

Action 中定义了对 DOM 的不同操作函数，并且每个函数都是 Dispatcher 中的函数，当 Action 接收到动作之后，会将该动作的相关信息告诉 Dispatcher。

（3）doDispatcher.js 用于导入 Flux 的 Dispatcher。

```
'use strict';
import {Dispatcher} from 'flux';
export default new Dispatcher();
```

（4）定义 Store。TodoStore.js 代码如下：

```
/**
 * Copyright (c) 2014-present, Facebook, Inc.
 * All rights reserved.
 *
 * This source code is licensed under the BSD-style license found in the
 * LICENSE file in the root directory of this source tree. An additional grant
 * of patent rights can be found in the PATENTS file in the same directory.
 */
'use strict';
import Counter from './Counter';
import Immutable from 'immutable';
import {ReduceStore} from 'flux/utils';
import Todo from './Todo';
import TodoActionTypes from './TodoActionTypes';
import TodoDispatcher from './TodoDispatcher';
class TodoStore extends ReduceStore {
  constructor() {
    super(TodoDispatcher);
  }
  getInitialState() {
    return Immutable.OrderedMap();
  }
  reduce(state, action) {
    switch (action.type) {
      case TodoActionTypes.ADD_TODO:
```

```
        // Don't add todos with no text.
        if (!action.text) {
          return state;
        }
        const id = Counter.increment();
        return state.set(id, new Todo({
          id,
          text: action.text,
          complete: false,
        }));
      case TodoActionTypes.DELETE_COMPLETED_TODOS:
        return state.filter(todo => !todo.complete);
      case TodoActionTypes.DELETE_TODO:
        return state.delete(action.id);
      case TodoActionTypes.EDIT_TODO:
        return state.setIn([action.id, 'text'], action.text);
      case TodoActionTypes.TOGGLE_ALL_TODOS:
        const areAllComplete = state.every(todo => todo.complete);
        return state.map(todo => todo.set('complete', !areAllComplete));
      case TodoActionTypes.TOGGLE_TODO:
        return state.update(
          action.id,
          todo => todo.set('complete', !todo.complete),
        );
      default:
        return state;
    }
  }
}
export default new TodoStore();
```

Store 对数据进行维护后，调用对 state 进行更新，然后重新渲染视图，效果如图 7-3 所示。



图 7-3　TodoList 运行效果图



该示例的源码没有带 node_modules 目录，在运行时如果出错，就先使用 npm install 下载依赖。

7.1.2　Mobx

Mobx 是一个用于状态管理的库，和 React 是一对很搭的组合。React 在 View 层提供了很好的解决方案，Mobx 对状态管理具有极大的简易性和可扩展性，二者搭配，能够很好地管理一些大型前端项目。本小节将介绍 Mobx 的原理。

Mobx 的工作原理大概如图 7-4 所示。在整个数据流中，首先用户的触发事件到达 Actions 中，然后依据事件属性及事件要求在 State 中对状态值进行修改，接下来用新的 State 数据计算所需要的 Computed values，最后响应渲染 UI 视图层。

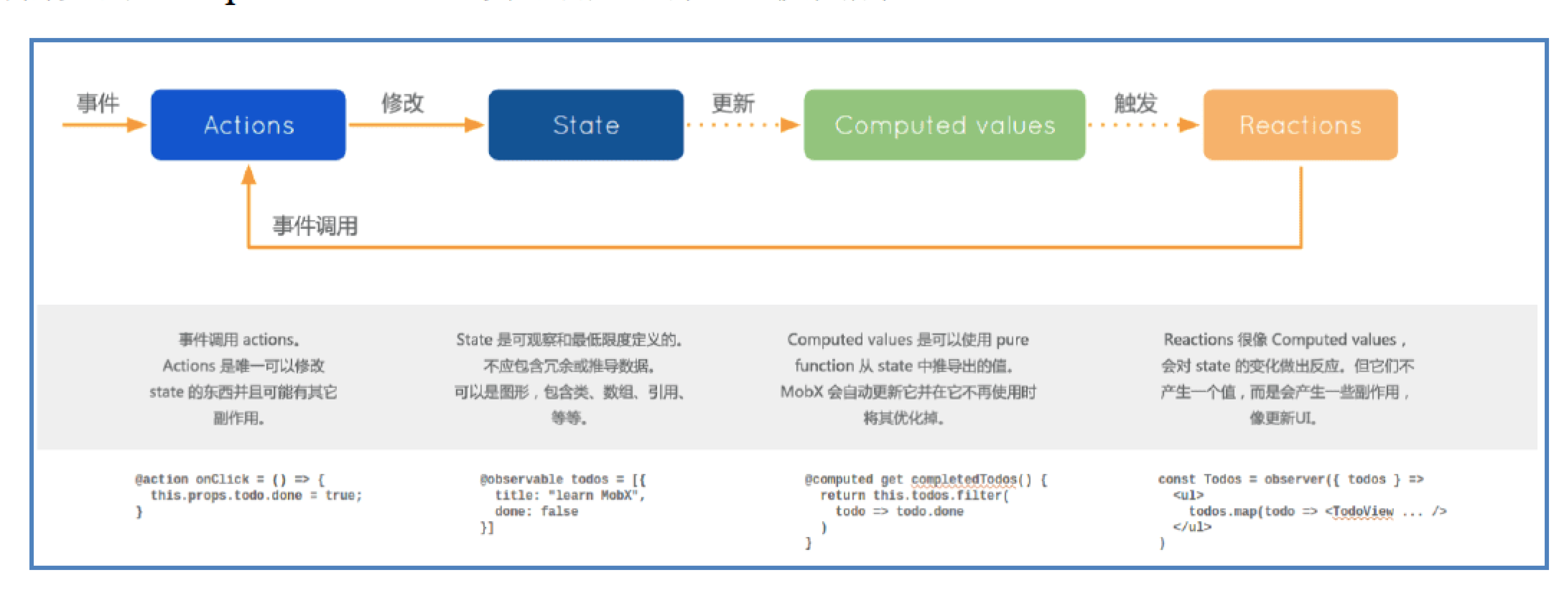


图 7-4　Mobx 原理图

7.1.3　Redux 应运而生

React 对 DOM 进行抽象管理，可以使用组件构建虚拟 DOM，组件的 HTML 结构不再是直接生成 DOM，而是映射生成虚拟的 JavaScript DOM 结构。React 通过 diff 算法将最小变更写入 DOM 中，从而减少 DOM 的实际次数，提升性能。React 只负责试图进行组件化管理，不涉及任何的数据和控制，但是在实际应用开发过程中一定会面临组件之间如何通信的问题，以及数据与视图之间如何实现便捷的数据流转的问题，需要数据流才能完成完整的应用。

Flux 与传统的 MVC 不同的是采用单向数据流，不允许 Model 和 Controller 互相引用。但是，Flux 也有其缺点。

- 首先，使用 Flux 时，同一个应用中可以同时使用多个 Store，并且不同的 Store 之间可能有依赖关系或引用关系，这样会造成系统的耦合度过高，后期维护困难。
- 其次，在 Flux 中，Store 封装了数据和数据处理逻辑，无法做到时间旅行，即让应用程序切换到任意时间的状态。

Redux 由 Flux 演变而来，但受 Elm 的启发，避开了 Flux 的复杂性。Redux 吸收了 Flux

的优点，例如单向数据流、依赖变动等特性。但是，Redux 还加入了一些新的特性，例如 undo、redo 等。同时，Redux 保持轻量级 API，便于实现更高层次的抽象。

7.2 Redux 核心概念

Redux 的核心作用是用于管理复杂的 JavaScript 应用中的数据状态，即我们常说的 state。这些状态包括 UI 状态、选中的标签、是否显示 Loading 动效、分页器、缓存数据等。state 本身是普通对象，没有修改器方法（setter 方法），而 state 的修改是先通过发起 action 描述当前发生了什么，最终通过 reducer 函数把 action 和 state 串起来。

7.2.1 store

Redux 的一个显著特点是整个应用中只提供一个 store。在 Redux 中，整个应用中的所有 state 均存储于同一棵对象树中，即 store。可以通过 reducer 创建 store，例如：

```
import { createStore } from 'redux'
import readApp from './reducer'
let store = createStore(readApp)
```

通常，我们在开发应用时，常常需要让服务端与客户端的 state 在结构上保持一致。在这种情况下，我们可以为 state 设置初始状态，在客户端使用服务端的 state 初始化本地数据，例如：

```
import { createStore } from 'redux'
const store = createStore(readApp, window.STATE_SERVER)
```

我们可通过 getState 方法获取当前的 state 数，如示例 7-2 所示。

【示例 7-2　getState 方法】

```
console.log(store.getState())
/* 输出
{
  status: 'learning',
  todos: [
    {
      text: 'learn store',
      completed: true,
    },
    {
      text: 'learn action',
      completed: false
    },
```

```
        {
          text: 'learn reducer',
          completed: false
        }
      ]
    }
    */
```

store 管理着整个应用的状态。store 提供了一个 dispatch 方法，可使用 dispatch 方法发送动作以修改 store 中的状态。随后可以再次通过 getState 方法重新获取最新的状态（state）。state 发生变化时，可以通过 store.subscribe(listener)注册监听器，以便在 state 变化时做相应的处理，例如：

```
// 打印初始状态
console.log(store.getState())

// 每次 state 更新时，打印日志
// 注意，subscribe() 返回一个函数用来注销监听器
const unsubscribe = store.subscribe(() =>
  console.log(store.getState())
)
```

完整的示例代码如下：

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'
import Counter from './components/Counter'
import counter from './reducers'

const store = createStore(
  counter,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__()
) // createStore 接收 3 个参数：reducer, preloadedState, enhancer
const rootEl = document.getElementById('root')
// 打印初始状态
console.log(store.getState().result)
// 每次 state 更新时，打印日志
// 注意，subscribe() 返回一个函数用来注销监听器
const unsubscribe = store.subscribe(() =>
  console.log(store.getState())
)
const render = () => ReactDOM.render(
  <Counter
```

```
    value={store.getState().result}
    onIncrement={() => store.dispatch({ type: 'INCREMENT' })}
    onDecrement={() => store.dispatch({ type: 'DECREMENT' })}
  />,
  rootEl
)

render()
store.subscribe(render)
```

运行结果如图 7-5 所示。

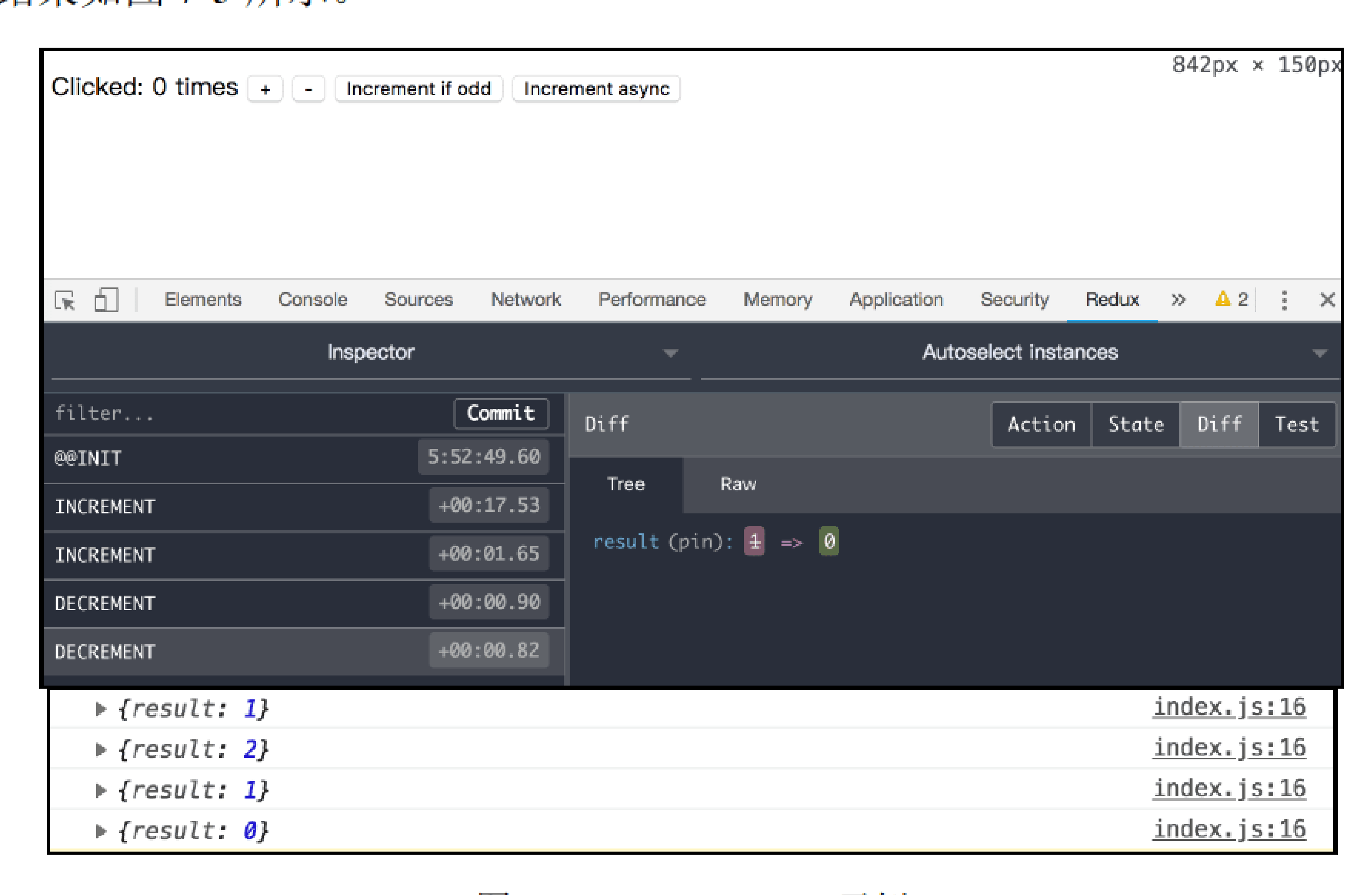


图 7-5　store.subscribe 示例

7.2.2　action

由于 state 是只读的，唯一可以改变状态的途径是触发 action。action 是一个普通对象，可用于描述已经发生的事件。例如，用户单击按钮、输入表单、拖拽、服务器数据推送、路由变化等，最终都会转换成对应的 action，而且这些 action 会按顺序执行，这种简单化的方法用起来非常方便。我们可以通过 store.dispatch(action)方法把 action 对象传递给 store，例如：

```
const action = {
    type: 'READ',
    msg: 'Keep on'
};
store.dispatch(action);
```

从代码可以看出，action 本质上是一个普通对象，其中的 type 属性是必需的，表示 action 的名称，用于描述将要执行的具体动作。通常 type 属性是字符串常量。action 中除了 type 字

段之外，其他结构开发者自行定义即可。

在实际的应用中，通常使用 action 创建函数来生成 action。在 Redux 中，action 创建函数只是简单地返回一个 action，例如：

```
function addToRead(text) {
  return {
    type: 'READ',
    msg: 'Keep on',
    index: 1
  }
}
```

在 Redux 中，把 action 创建函数返回的结果传递给 dispatch 方法即可触发一次 dispatch，例如：

```
dispatch(addToRead('books'))
```

使用 action 创建函数的优点是便于移植和测试具体的业务单元，同时也可以使用异步非纯函数作为 action 的创建函数，形成异步控制流。state 只能通过 action 触发修改，不允许视图或网络请求直接修改 state，只能通过视图或网络请求描述需要发生的变更。最终，所有的修改都被集中处理，严格按照顺序依次执行。

完整代码如下：

```
/*
 * action 类型
 */

export const ADD_READ = 'ADD_READ';
export const TOGGLE = 'TOGGLE'
export const DELETE_READ = 'DELETE_READ'

/*
 * 其他的常量
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * action 创建函数
 */
```

```
export function addRead(text) {
  return { type: ADD_READ, text }
}

export function toggle(index) {
  return { type: TOGGLE, index }
}

export function deleteRead(filter) {
  return { type: DELETE_READ, filter }
}
```

7.2.3 reducer

7.2.2 小节中提到 action 描述发生的具体动作，但是 action 并没有描述如何更新应用的 state。通过 dispatch 发起 action 之后，最终是通过 reducer 指定如何修改 state tree 的。也就是说，reducer 是用来修改状态的。我们先确定 state 对象的结构，再开始编写 reducer。reducer 是纯函数，接收 action 和当前 state 作为参数，并返回一个新的 state。

```
/* state 结构
{
  status: 'learning',
  todos: [
    {
      text: 'learn store',
      completed: true,
    },
    {
      text: 'learn action',
      completed: false
    },
    {
      text: 'learn reducer',
      completed: false
    }
  ]
}
*/
const reducer = function (state = [], action) {
  // 其他逻辑处理可以写在这里
  switch (action.type) {
    // ADD_READ 操作
```

```
      case 'ADD_READ':
        return [
          ...state,
          {
            text: action.text,
            completed: false
          }
        ]
      case 'COMPLETE_READ':
        return state.map((articles, index) => {
          if (index === action.index) {
            return Object.assign({}, articles, {
              completed: true
            })
          }
          return articles
        })
      default:
        return state
    }
  };
```

特别需要注意的是，不要在 reducer 中直接修改传入的 state，可以使用 Object.assign()或延展符号返回新的 state。我们需要修改数组中指定的数据项而又不希望导致突变时，推荐的做法是在创建一个新的数组后将那些无须修改的项原封不动移入，接着对需要修改的项用新生成的对象替换。同时，在 default 情况下返回旧的 state 即可。此时，重新通过 store.getState 返回新的状态，可以看出前后两次状态有可能发生变化。最终通过 store.subscribe，以 render 作为参数，一旦监听到 state 发生改变，就执行 render 函数，这样就可以触发视图更新。

整体示例代码如下：

```
  import { combineReducers, createStore } from 'redux'
  let reducer = combineReducers({ visibilityFilter, Count })
  let store = createStore(reducer)

  const reducer = function (state = [], action) {
    //其他逻辑处理可以写在这里
    switch (action.type) {
      case 'ADD_READ':
        return [
          ...state,
          {
            text: action.text,
            completed: false
```

```
        }
      ]
    case 'COMPLETE_READ':
      return state.map((articles, index) => {
        if (index === action.index) {
          return Object.assign({}, articles, {
            completed: true
          })
        }
        return articles
      })
    default:
      return state
  }
};


function addReadingBooks(state = [], action) {
   // do something...
   return nextState
 }

 function showReadingBooks(state = [], action) {
   //其他逻辑处理可以写在这里
   //处理完毕之后返回新的状态
   return nextState
 }

 let App = combineReducers({
   addReadingBooks,
   showReadingBooks
 })
```

7.2.4 connect

Redux 和 React 之间没有关系。Redux 支持 React、Angular、Ember、jQuery 甚至纯 JavaScript。尽管如此，Redux 最好还是和 React 和 Deku 这类框架搭配起来用，因为这类框架允许你以 state 函数的形式来描述界面，Redux 可以通过 action 的形式来发起 state 变化。

React-Redux 提供 connect 方法，用于从 UI 组件生成容器组件。connect 的意思就是将这两种组件连起来。

```
import { connect } from 'react-redux';
const Count = connect()(List);
```

在上面的代码中，Count 是一个 UI 组件，List 是由 React-Redux 通过 connect 方法自动生成的容器组件。只是这样纯粹地把 Memos 包裹起来毫无意义，完整的 connect 方法可以这样使用：

```
import { connect } from 'react-redux'
const List = connect(
  mapStateToProps
)(Count)
```

在上面的代码中，connect 方法接收两个参数：mapStateToProps 和 mapDispatchToProps。它们定义了 UI 组件的业务逻辑。前者负责输入逻辑，即将 state 映射到 UI 组件的参数（props）；后者负责输出逻辑，即将用户对 UI 组件的操作映射成 action。

7.2.5　总结

Redux 的核心设计是使用单向数据流。这种设计使得在所有的应用中数据都将遵循相同的生命周期，在应用开发过程中将更有利于预测和理解应用的状态。除此之外，单向数据流的优势还体现在避免在相同应用中出现重复的、不可互相引用的数据，使得数据更加单纯。Redux 的数据流转如图 7-6 所示。

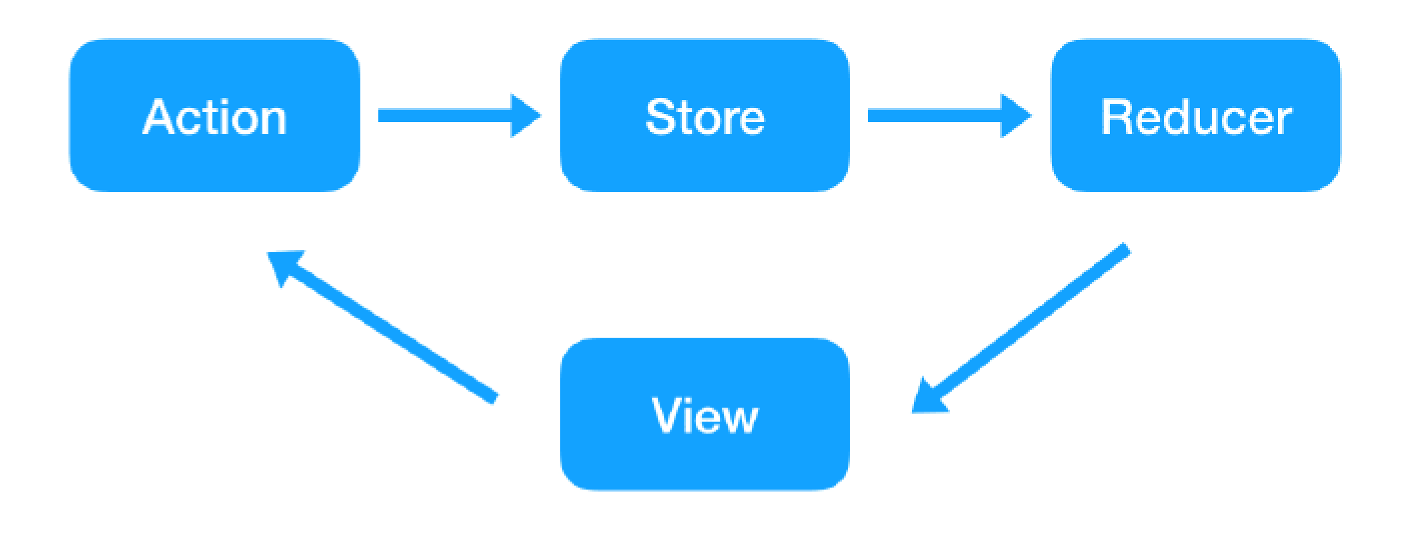


图 7-6　Redux 的数据流转

Redux 应用中的数据遵循如下生命周期：

首先，调用 store.dispatch(action)。可以在任何地方调用 store.dispatch(action)，例如可以在组件中、单击事件回调中、异步请求的回调中、定时器中调用，其中 action 是用于描述当前具体发生事件的对象。

其次，store 调用传入的 reducer 函数。store 调用 reducer 时，会向对应的 reducer 传入当前的 state tree 和 action。reducer 通过 action 的类别和当前的 state 进行计算，最终返回一个新的 state。reducer 是纯函数，每次传入相同的输入必须产生相同的输出。

再次，把多个 reducer 的输出合并成单一的 state tree。Redux 提供的 combineReducers 辅助函数可以把根 reducer 拆分成多个函数，用于分别处理 state 树的一个分支。例如：

```
function addReadingBooks(state = [], action) {
   //其他逻辑处理可以写在这里
   return nextState
 }
```

```
function showReadingBooks(state = [], action) {
  //其他逻辑处理可以写在这里
  return nextState
}

let App = combineReducers({
  addReadingBooks,
  showReadingBooks
})
```

combineReducers 会把两个 state 合并成一个 state tree。

最后，store 保存了最新的完整的 state tree。可以调用 store.getState()获取当前 state，也可以根据最新的 state 更新组件的 View。

7.3 Redux 生态

随着 Redux 广泛使用，衍生出了丰富工具集和可扩展的 Redux 生态系统，包括中间件、工具库、库和插件等，足以维护大型项目。本节就 Redux 中常用的插件做简要介绍。

7.3.1 redux middleware

Redux 允许开发者自定义中间件来处理影响 store 的 dispatch 逻辑，也就是说，Redux 的中间件主要用在 store 的 dispatch 函数上。dispatch 函数的作用是发送 actions 给一个或多个 reducer 来影响应用状态。中间件可以增强默认的 dispatch 函数。

Redux 中间件被设计成可组合的、会在 dispatch 方法之前调用的函数，例如 redux-logger 中间件（下一小节会详细介绍）。

7.3.2 redux-logger

redux-logger 能够对所有 action 发生后生成的 state 进行记录，即日志中间件。在项目调试阶段，我们可以根据 console.log 输出判断业务中具体 state 变化。redux-logger 使用示例如下：

安装 redux-logger：

```
npm install --save redux-logger
```

在项目中使用 redux-logger：

```
import thunk from 'redux-thunk';
import promise from 'redux-promise';
import createLogger from 'redux-logger';
```

```
// 实例化
const logger = createLogger();
// 应用中间件
const createStoreWithMiddleware = applyMiddleware(thunk, promise,
logger)(createStore);
// 连接到 store
const store = createStoreWithMiddleware(reducer);
```

以上示例代码中，首先导入 redux-logger，运行 createLogger 方法，将返回结果赋值给常量。然后将 looger 传入 applyMiddleware()，传入 createStore 方法，就完成了 store.dispatch()的功能增强。applyMiddleware 方法是 Redux 的原生方法，其作用是将所有中间件组成一个数组，依次执行。applyMiddleware 方法的三个参数就是三个中间件，其中 redux-logger 一定要放在最后，否则输出结果会发生错误。

使用 redux-logger 时，直接调用 createLogger()，不传入任何参数则使用默认的配置。redux-logger 提供的可配置参数如下：

```
{
  predicate,      // 限制 logger 的条件
  collapsed,      //分组，可以是 Boolean 值或函数，该函数入参为 getState 函数和 action 参数
  duration = false: Boolean,     // 打印每个 action
  timestamp = true: Boolean,     // 打印每个 action 执行的时间戳

  level = 'log': 'log' | 'console' | 'warn' | 'error' | 'info',//console 的级别
  colors: ColorsObject, //为 title、上一个 state、action 和下一个 state 定义不同的颜色
  titleFormatter,            // 格式化标题

  stateTransformer,          // state 转换
  actionTransformer,         // action 转换
  errorTransformer,                  // error 转换

  logger = console: LoggerObject,    // 'console' API 的实现
  logErrors = true: Boolean,         // 记录 catch、log 和 re-throw errors

  diff = false: Boolean,             // 标注 states 之间的差异
  diffPredicate                      // 标注 states 之间的差异的条件
}
```

通常，可以在开发环境中开启日志，示例代码如下：

```
const middlewares = [];

if (process.env.NODE_ENV === 'development') {
  const { logger } = require('redux-logger');
```

```
  middlewares.push(logger);
}

const store = compose(applyMiddleware(...middlewares))(createStore)(reducer);
```

过滤指定的 action 类型：

```
createLogger({
  predicate: (getState, action) => action.type !== AUTH_REMOVE_TOKEN
});
```

定义 stateTransformer 方法：

```
import { Iterable } from 'immutable';

const stateTransformer = (state) => {
  if (Iterable.isIterable(state)) return state.toJS();
  else return state;
};

const logger = createLogger({
  stateTransformer,
});
```

使用配置执行日志批量处理示例代码如下：

```
import { createLogger } from 'redux-logger';

const actionTransformer = action => {
  if (action.type === 'BATCHING_REDUCER.BATCH') {
    action.payload.type = action.payload.map(next => next.type).join(' => ');
    return action.payload;
  }

  return action;
};

const level = 'info';

const logger = {};
// 日志处理
for (const method in console) {
  if (typeof console[method] === 'function') {
    logger[method] = console[method].bind(console);
  }
}
```

```
// 打印日志方法
logger[level] = function levelFn(...args) {
  const lastArg = args.pop();

  if (Array.isArray(lastArg)) {
    return lastArg.forEach(item => {
      console[level].apply(console, [...args, item]);
    });
  }

  console[level].apply(console, arguments);
};

export default createLogger({
  level,
  actionTransformer,
  logger
});
```

使用 redux-logger 可以在控制台输出 action 的信息，所以首先要获取前一个 action、当前 action，然后是下一个 action，打印出的日志如图 7-7 所示。

```
▸ {result: 1}                                          index.js:21
▾ action INCREMENT @ 18:03:05.151                      redux-logger.js:1
    prev state ▸ {result: 0}                           redux-logger.js:1
    action     ▸ {type: "INCREMENT"}                   redux-logger.js:1
    next state ▸ {result: 1}                           redux-logger.js:1
▸ {result: 0}                                          index.js:21
▾ action DECREMENT @ 18:03:06.848                      redux-logger.js:1
    prev state ▸ {result: 1}                           redux-logger.js:1
    action     ▸ {type: "DECREMENT"}                   redux-logger.js:1
    next state ▸ {result: 0}                           redux-logger.js:1
▸ {result: 1}                                          index.js:21
▾ action INCREMENT @ 18:03:07.647                      redux-logger.js:1
    prev state ▸ {result: 0}                           redux-logger.js:1
    action     ▸ {type: "INCREMENT"}                   redux-logger.js:1
    next state ▸ {result: 1}                           redux-logger.js:1
```

图 7-7　redux-logger 示例

7.3.3　redux-thunk

redux-thunk 是用于在 redux 中处理异步 action 的中间件，异步 action 的场景包括需要在 action 中执行 setTimeout，以及需要通过 fetch 方法调用服务端 API 等，对于异步 action 的场景可以使用 redux-thunk 中间件来优化代码流程。redux-thunk 统一了异步和同步 action 的调用方式，使得异步过程放在 action 级别解决，避免异步操作对 component 的耦合。例如：

```
store.dispatch({ type: 'ADD', text: '添加完成' })
setTimeout(() => {
  store.dispatch({ type: 'COMPLETE'})
```

```
}, 1000)
```

这段代码中使用 setTimeout 延迟 1 秒触发 SUM 操作。使用 redux-thunk 中间件进行改造，首先安装 redux-thunk：

```
npm install redux-thunk --save-dev
```

然后在代码中使用 redux-thunk:

```
import thunk from 'redux-thunk
import {createStore,applyMiddleware} from 'redux'

import counter from './reducers/counter'
let store = createStore(counter, applyMiddleware(thunk))
```

使用 redux-thunk 中间件，可以让 action 创建函数先不返回一个 action 对象，而是返回一个函数，函数传递两个参数(dispatch, getState)，我们可以在函数体内进行业务逻辑的封装：

```
function add() {
   return {
      type: 'ADD',
   }
}

function addIfOdd() {
   return (dispatch, getState) => {
      const currentValue = getState();
      if (currentValue % 2 == 0) {
         return false;
      }
      dispatch(add())
   }
}
```

完整代码如下：

```
'use strict';
import {createStore, applyMiddleware} from 'redux';
import thunk from 'redux-thunk';

// count 的操作集合
function count(state = 0, action) {
   switch (action.type) {
      case 'ADD':
         return state + 1;
      case 'REDUCER':
         return state - 1;
```

```
        default:
            return state;
    }
}
const store = createStore(count,applyMiddleware(thunk));
//action 创建函数
function add() {
    return {
        type: 'ADD',
    }
}
function reducer() {
    return {
        type: 'REDUCER'
    }
}
// 奇数求和方法
function addIfOdd() {
    return (dispatch, getState) => {
        const currentValue = getState();
        if (currentValue % 2 == 0) {
            return false;
        }
        dispatch(add())
    }
}
// 异步求和方法
function addAsy(delay = 2000) {
    return (dispatch, getState) => {
        setTimeout(() => {
            dispatch(add())
        }, delay)
    }
}

//获取当前值
let currentValue = store.getState();
//创建一个监听
store.subscribe(() => {
    const previosValue = currentValue;
    currentValue = store.getState();
    console.log('上一个值:', previosValue, '当前值:', currentValue)
});
```

```
//分发任务
store.dispatch(add());
store.dispatch(add());
store.dispatch(add());
store.dispatch(add());
store.dispatch(reducer());
store.dispatch(addIfOdd());
store.dispatch(addAsy());
```

redux-thunk 中间件可以在发出 action 到 reducer 函数接受 action 之前执行具有副作用的异步操作。redux-thunk 源码的实现也较为简单：

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}
// thunk 中间件
const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

判别 action 的类型，如果 action 是函数，就调用这个函数，实参为 dispatch 和 getState，因此我们在定义 action 为 thunk 函数时，一般形参为 dispatch 和 getState。

通过 redux-thunk 来处理异步操作，thunk 仅仅做了执行这个函数，并不在乎函数主体内是什么，也就是说 thunk 使得 redux 可以接受函数作为 action，但是函数的内部可以多种多样。action 如此繁衍开来、异步操作太为分散、规范缺失，不利于后期维护。action 是一个 async 函数，需要约定一些异步操作时所遵循的规则，即使不能集中展示所有的异步操作，但是通过约定可以减少函数 action 的复杂度：

```
 export default function(){
  return  function(dispatch){
    await result=fetch(...)
    result.then(...)
  }
};
```

7.3.4　redux-saga

redux-saga 也是 Redux 的一个中间件，与 redux-thunk 功能类似，也是主要集中管理 React 应用中的异步操作。redux-saga 最大的特点是使用 generator(ES6)的形式，采用监听的形式进行工作，可以使用同步的方式编写异步代码，使得项目流程拆分更细，应用的逻辑和视图拆分更加清晰，分工明确。项目中异步的 action 和复杂逻辑的 action 都可以放到 redux-saga 中去处理，使得模块逻辑更加清爽。

首先，安装 redux-saga：

```
npm install --save redux-saga
```

在项目中使用 redux-saga 时，引入 redux-saga 并连接到 store 中：

```
import React from 'react'
import ReactDOM from 'react-dom'
import {
  createStore,
  applyMiddleware
} from 'redux'
import logger from 'redux-logger'
import createSagaMiddleware from 'redux-saga'
// 引入相关组件
import Counter from './components/Counter'
import counter from './reducers/counter'
import mySaga from './sagas'
// 创建 saga 中间件
const sagaMiddleware = createSagaMiddleware()
// 创建 store
// createStore 接受 3 个参数：reducer, preloadedState, enhancer
const store = createStore(
  counter,
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__(),
  applyMiddleware(sagaMiddleware, logger)
)

// 运行 saga
sagaMiddleware.run(mySaga)

const rootEl = document.getElementById('root')
// 打印初始状态
console.log(store.getState().result)
// 每次 state 更新时，打印日志
// 注意 subscribe() 返回一个函数用来注销监听器
```

```
const unsubscribe = store.subscribe(() =>
  console.log(store.getState())
)
// 渲染方法
const render = () => ReactDOM.render(
  <Counter
    value={store.getState().result}
    onIncrement={() => store.dispatch({ type: 'INCREMENT' })}
    onDecrement={() => store.dispatch({ type: 'DECREMENT' })}
    onSomeButtonClicked= {() =>
      store.dispatch({type: 'USER_FETCH_REQUESTED', payload: {userId:
'test'}})
    }
  />,
  rootEl
)
// 调用渲染方法
render()
store.subscribe(render)
```

使用 createSagaMiddleware 方法创建 redux-saga 的 Middleware，然后在创建的 redux 的 store 时，使用 applyMiddleware 函数将创建的 saga Middleware 实例绑定到 store 上，最后调用 saga Middleware 的 run 函数来执行某个或者某些 Middleware。

使用 redux-saga，我们首先生成一个集中处理异步的 sagas.js 文件。sagas.js 文件代码示例如下：

```
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from './api'

// 通过 USER_FETCH_REQUESTED action 触发
function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "USER_FETCH_FAILED", message: e.message});
   }
}

/*
  每次执行`USER_FETCH_REQUESTED` 操作时调用 fetchUser
  Allows concurrent fetches of user.
*/
```

```
function* mySaga() {
  yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/*
  也可以使用 takeLatest 方法实现
*/
function* mySaga() {
  yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}

export default mySaga;
```

简单来说，redux-saga 相当于在 Redux 原有的数据流中增加了一层监控，捕获监听到的 action 并进行处理之后，添加一个新的 action 到相应的 reducer 中执行处理流程。在 redux-saga 中的 action 与在 redux 中同步 action 的样式是相同的。

redux-saga 是通过 ES6 中的 generator 实现的，本质是一个可以自执行的 generator。在 saga 中，可以使用 takeEvery 或者 takeLatest 等 API 来监听 action。当某个 action 触发后，saga 可以使用 call、fetch 等 API 方法发起异步操作，操作完成后使用 put 函数触发 action，同步更新 state，从而完成整个 state 的更新。

在 redux-saga 中，所有的任务都必须通用 yield effect 来完成，例如：

```
import { call, put, select , take} from 'redux-saga/effects';

function* mySaga() {
  yield takeEvery ("USER_FETCH_REQUESTED", fetchUser);
}
```

redux-saga 中定义了 effect，effect 本质就是一个特定的函数，返回的是纯文本对象。也就是说，通过 effect 函数会返回一个字符串，saga-middleware 根据这个字符串来执行真正的异步操作。redux-saga 常见的 effect 有 call、put、select、takeEvery、takeLatest、take、all 等。

redux-saga 中使用 call 用来调用异步函数，将异步函数和函数参数作为 call 函数的参数传入，最终该函数返回一个对象。call 方法的主要作用是方便测试，同时也能让我们的项目代码更加规范化。同 JavaScript 原生的 call 方法类似，call 函数也可以指定 this 对象，只需要把 this 对象当第一个参数传入 call 方法即可。saga 同样提供 apply 函数，作用同 call 一样，参数形式同 JavaScript 原生 apply 方法。

```
import { call, put, select , take} from 'redux-saga/effects';

export function* getAdData(url) {
    yield put({type: START_FETCH});
    yield  delay(delayTime);
    try {
```

```
        return yield call(get,url);
    } catch (error) {
        yield put({type: FETCH_ERROR})
    }finally {
        yield put({type: FETCH_END})
    }
}
// 异步获取数据方法
export function* getAdDataFlow() {
    while (true){
        let request = yield take(GET_AD);
        let response = yield call(getAdData, url);
        yield put({type: GET_AD_RESULT_DATA, data:response.data})
    }
}
```

在 redux-saga 中使用 put 触发 action，其作用类似于 react 中的 dispatch：

```
import { call, put, select , take} from 'redux-saga/effects';
// 异步获取用户信息
function* fetchUser(action) {
  const user = 'test';
  try {
    // const user = yield call(Api.fetchUser, action.payload.userId);
    yield put({type: "USER_FETCH_SUCCEEDED", user: user});
  } catch (e) {
    yield put({type: "USER_FETCH_FAILED", message: e.message});
  }
}
```

redux-saga 中的 select 方法的作用与 getState 的作用类似：

```
import { call, put, select , take} from 'redux-saga/effects';

// 加载用户信息
export function* loadUserInfo() {
  try {
    yield take('FETCH_USER_SUCCESS');

    const user = yield select(getUserFromState);

    yield put({type: 'FETCH_DEPARTURE3_SUCCESS', departure});
  } catch(error) {
    yield put({type: 'FETCH_FAILED', error: error.message});
  }
```

```
}
```

takeEvery 用来监听 action，每个 action 都触发一次，如果其对应的是异步操作，那么每次都会发起异步请求，而不论上次的请求是否返回，例如：

```
import { call, put, select , take} from 'redux-saga/effects';

function* mySaga() {
  yield takeEvery ("USER_FETCH_REQUESTED", fetchUser);
}
```

与 takeEvery 相对应的方法有 takeLatest。takeLatest 的作用与 takeEvery 一样，唯一的区别是它只关注最后一次，也就是最近一次发起的异步请求，如果上次请求还未返回，就会被取消。使用示例：

```
import { call, put, select , take} from 'redux-saga/effects';

function* watchFetchData() {
  yield takeLatest('FETCH_REQUESTED', fetchData)
}
```

take 的表现与 takeEvery 一样，都是监听某个 action，但与 takeEvery 不同的是，take 不是每次 action 触发的时候都响应，而只是在执行顺序执行到 take 语句时才会响应 action。也就是说，take 只有在执行流达到时才会响应对应的 action，而 takeEvery 则一经注册，就会响应 action。当在 genetator 中使用 take 语句等待 action 时，generator 被阻塞，等待 action 被分发，然后继续往下执行。而 takeEvery 只是监听每个 action，然后执行处理函数。对于何时响应 action 和 如何响应 action，takeEvery 并没有控制权。而使用 take 则不一样，我们可以在 generator 函数中决定何时响应一个 action，以及一个 action 被触发后做什么操作，例如：

```
import { call, put, select , take} from 'redux-saga/effects';

export function* getAdDataFlow() {
    while (true){
        let request = yield take(homeActionTypes.GET_AD);
        let response = yield call(getAdData,request.url);
        yield
put({type:homeActionTypes.GET_AD_RESULT_DATA,data:response.data})
    }
}
```

all 提供了一种并行执行异步请求的方式。之前介绍过执行异步请求的 api 中大都是阻塞执行，只有当一个 call 操作放回后，才能执行下一个 call 操作。call 提供了一种类似 Promise 中的 all 操作，可以将多个异步操作作为参数传入 all 函数中，如果其中有一个 call 操作失败或者所有 call 操作都成功返回，则本次 all 操作执行完毕，示例如下：

```
import { all, call } from 'redux-saga/effects'

// correct, effects will get executed in parallel
const [users, repos]  = yield all([
  call(fetch, '/users'),
  call(fetch, '/repos')
])
```

redux-saga 集中处理了所有的异步操作，项目中的异步接口非常清晰。在 redux-saga 中的 action 是普通对象，与 Redux 中同步的 action 完全一致，通过 worker 和 watcher 可以实现非阻塞异步调用，并且同时可以实现非阻塞调用下的事件监听，异步操作的流程是可以控制的，可以随时取消相应的异步操作。

7.4 Redux 进阶

7.4.1 理解 middleware 原理

Redux 主要输出 createStore、combineReducers、bindActionCreators、applyMiddleware、compose 五个接口。本节中，我们重点讨论 middleware 的核心原理。

Redux 的核心是控制和管理所有的数据输入与输出，使用纯函数 dispatch 派发 action 来更改数据，其功能简单且固定。middleware 允许我们 dispatch 一个 action 之后，在到达 reducer 之前先做一些额外的处理，可以理解为每一个 middleware 都在增强 dispatch 的功能。

例如，当需要记录每次 dispatch 的 action 时，最简单的方法是在每次 dispatch 之前直接加上 log 日志，代码如下：

```
let action = ADD('2');
console.log('dispatch', action);
// 触发事件
store.dispatch(action);
console.log('next state', store.getState())
```

为了方便在项目中多处使用，可封装成统一的函数方法：

```
export default function markDispatch (store, action) {
    console.log('dispatch', action);
    store.dispatch(action);
    console.log('next state', store.getState())
}
```

然后在需要使用的地方引入该方法，并使用 markDispatch 替换原有的 dispatch：

```
import markDispatch from './utils'
```

但是我们并不想每次用的时候都 import 这个函数，所以我们需要使用新方法直接替代 dispatch:

```
const next = store.dispatch;
store.dispatch = function (action) {
   let action = ADD ('2');
   console.log('dispatch', action);
   next(action)
   console.log('next state', store.getState())
}
```

这样我们在用 dispatch 的时候就自动附带了使用 console.log 记录 action 和 next state 的功能，这里每一个附带上的功能就是一个 middleware，都是用来增强 dispatch 作用的。

我们进一步思考如何实现多个 middleware 的串联顺序执行。例如，在每次 dispatch 时记录 action 和 next state 的基础上，也需要记录每次 dispatch 错误的原因，那么执行的顺序是 dispatch→markDispatch→reportDispatchError，后面的中间件需要接收到前面改造后的 dispatch。

首先，对 dispatch 记录函数进行改造：

```
function markDispatch (store) {
  let next = store.dispatch;
  store.dispatch = dispatchAndLog(action) => {
    console.log('dispatching', action);
    let result = next(action);
    console.log('next state', store.getState());
    return result;
  }
}
```

直接返回 result 函数，可以在后面实现一个链式调用。继续改造，让每一个中间件函数接收一个 dispatch，然后返回一个改造后的 dispatch 来作为下一个中间件函数的 next：

```
const markDispatch = store => next => action => {
  console.log('dispatching', action)
  return next(action)
}
```

类似的，我们继续封装一个记录报错函数：

```
function reportDispatchError(store) {
  let next = store.dispatch;
  store.dispatch = (action) => {
    try {
      return next(action);
    } catch (err) {
```

```
      console.error('Caught an exception!', err);
      Goldlog.record(err, {
        extra: {
          action,
          state: store.getState();
        }
      })
      throw err;
    }
  }
}
```

改造后:

```
const reportDispatchError = store => next => action => {
  try {
    return next(action)
  } catch (err) {
      console.error('Caught an exception!', err);
      Goldlog.record(err, {
        extra: {
          action,
          state: store.getState();
        }
      })
      throw err;
    }
  }
}
```

接着，我们改造一下 applyMiddleware，接收一个 middlewares 数组：

```
export default function applyMiddleware(middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  let dispatch = store.dispatch
  middlewares.forEach(middleware =>
    dispatch = middleware(store)(dispatch)
  )
  return Object.assign({}, store, { dispatch })
}
```

查看 Redux 中 applyMiddleware 的源码，基本上类似于下面的情况：

```
export default function compose(...funcs) {
```

```
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}

// 可以看出 compose 的作用是上一个函数的返回结果作为下一个函数的参数传入
export default function applyMiddleware(...middlewares) {
  // 可以这么做是因为在 creatStore 里，当发现 enhancer 是一个函数的时候
  // 会直接 return enhancer(createStore)(reducer, preloadedState)

  return (createStore) => (...args) => {
    // 之后就在这里先建立一个 store
    const store = createStore(...args)
    let dispatch = store.dispatch
    let chain = []
    // 将 getState 跟 dispatch 函数暴露出去
    const middlewareAPI = {
      getState: store.getState,
      dispatch: (...args) => dispatch(...args)
    }
//这里返回 chain 的一个数组，里面装的是 wrapDispatchToAddLogging 那一层
// 相当于先给 middle 预处理，接下来只需要开始传入 dispatch 即可
    chain = middlewares.map(middleware => middleware(middlewareAPI))

    dispatch = compose(...chain)(store.dispatch)
    // wrapCrashReport(wrapDispatchToAddLogging(store.dispatch))
    // 此时返回了上一个 dispatch 的函数作为 wrapCrashReport 的 next 参数
    // wrapCrashReport(dispatchAndLog)
    // 最后返回最终的 dispatch
    return {
      ...store,
      dispatch
    }
  }
}
```

其中...middleware (arguments) 遵循 Redux middleware API 的函数。每个 middleware 接受

Store 的 dispatch 和 getState 函数作为命名参数，并返回一个函数。该函数会被传入被称为 next 的下一个 middleware 的 dispatch 方法，并返回一个接收 action 的新函数，这个函数可以直接调用 next(action)，或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store 的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数是({ getState, dispatch }) => next => action。

7.4.2 手动实现 middleware

7.4.1 小节中介绍了 applyMiddleware 的基本原理，了解到 middleware 的函数形式是({ getState, dispatch }) => next => action，本小节我们实现自定义中间件。

```
import { createStore, applyMiddleware } from 'redux'
import count from './reducers'

function logger({ getState }) {
  return next => action => {
    console.log('will dispatch', action)

    // 调用 middleware 链中下一个 middleware 的 dispatch
    let returnValue = next(action)

    console.log('state after dispatch', getState())

    // 一般会是 action 本身，除非后面的 middleware 修改了它
    return returnValue
  }
}

let store = createStore(
  count,
  ['ADD'],
  applyMiddleware(logger)
)

store.dispatch({
  type: 'SUM',
  text: ' perform sum…'
})
// 将打印如下信息
// will dispatch: { type: 'SUM', text: 'perform sum…' }
// state after dispatch: [ 'ADD', 'perform add…' ]
```

第 8 章

React架构

React 不仅仅是一个库，也不是一个框架，而是一个庞大的生态体系。若希望充分使用 React 尽可能多的特性，整个技术栈都要相应地配合改造，则我们要学习从后端到前端的一整套解决方案。因此，使用 React 时，合理的选择是采用 React 的整个技术栈。本章将开始详细分析如何搭建一个 React 应用架构，所以本章的例子希望读者使用 WebStrom 创建，或者使用脚手架创建，不再直接使用<script></script>引入 React。

8.1 文件结构

本节采用 React+Redux+React-Router+Less+ES6+webpack 介绍 React 实现一个完整应用的文件结构。React 的 CLI 脚手架工具 create-react-app 屏蔽了和 React 无关的配置，如 Babel、webpack。下面我们使用这个工具快速创建一个项目。

【示例 8-1　文件结构演示】

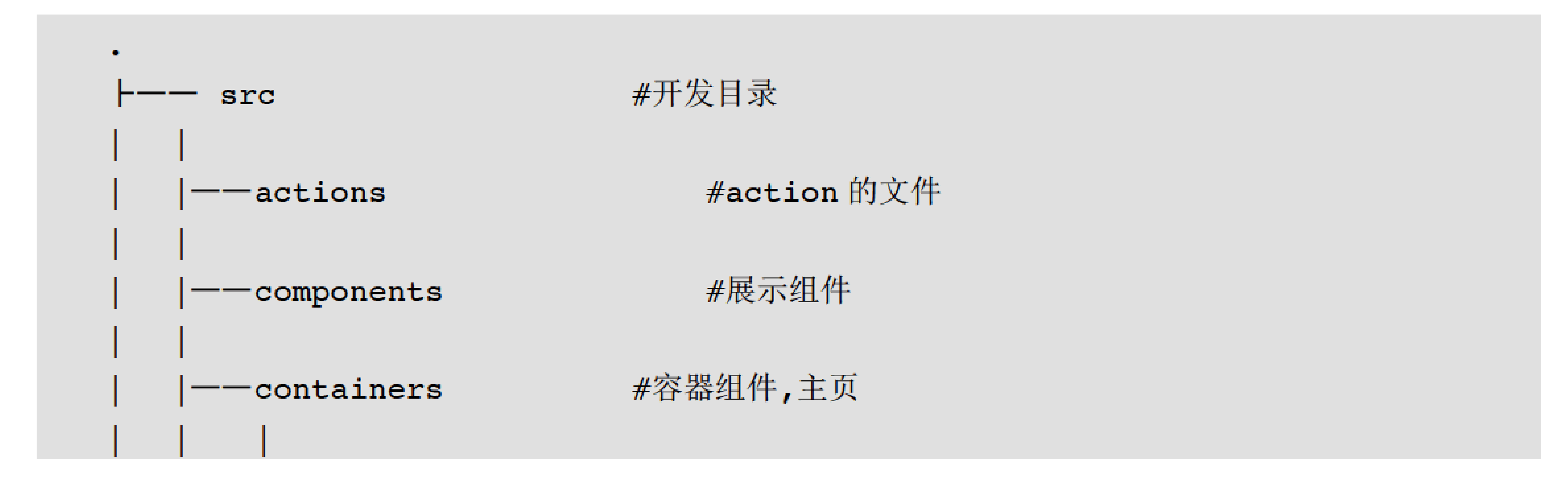
用 npm 安装一个全局的 create-react-app（如果前面没有安装过）：

```
npm install -g create-react-app
```

然后运行：

```
create-react-app hello-world
```

初始化了一份 React 项目，只要运行 npm run start 就能启动开发服务器了。例如：



```
.
├── src                      #开发目录
|   |
|   |——actions                   #action 的文件
|   |
|   |——components                #展示组件
|   |
|   |——containers            #容器组件,主页
|   |    |
```

```
|    |     |——Alert            #业务组件
|    |     |
|    |     |——Notification       #业务组件
|    |     |
|    |——reducers                 #reducer 文件
|    |
|    |—static               #静态文件
|    |
|    |——store               #store 配置文件
|    |
|    |——utils               #工具文件
|    |
|    |—index.less           #样式文件
|    |
|   └———index.js                 #入口文件
|
├——— public                      #发布目录
├——— node_modules                #包文件夹
├——— .gitignore
├——— .jshintrc
├——— webpack.production.config.js      #生产环境配置
├——— webpack.config.js             #webpack 配置文件
├——— package.json                      #环境配置
└——— README.md                         #使用说明
```

首先，以功能为目录。将 components、containers、stores 按其功能放在一个目录内，将组件都放在 components 目录内，containers 则是组装 component。

其次，组件扁平化结构。例如，可以将所有的组件都放在 components 目录，这种适合简单组件少或者比较单一的情况。

最后，以组件为目录。组件内需要的文件放在同一个目录下，例如 Alert 和 Notification 可以建两个目录，目录内部有代码、样式和测试用例。

8.2 CSS 方案

8.2.1 CSS Modules

CSS 的全称是 Cascading Style Sheets，即层叠样式表，是网页样式的一种描述方法。CSS 为 HTML 标记语言提供了一种样式描述，定义了其中元素的显示方式。严格来讲，CSS 不能算是一种编程语言，由于 ES2015/2016 的快速普及和 Babel/webpack 等工具的迅猛发展，CSS

的发展显得非常缓慢，逐渐成为大型项目工程化的痛点，变成前端走向彻底模块化前必须解决的难题。不论是用 jQuery 还是 React 开发，都会遇到一系列的 CSS 问题：

- 全局污染
- 命名混乱
- 依赖引入复杂
- 无法共享变量
- 代码冗余

为了让 CSS 也能适用软件工程方法解决上述列举的问题，从近些年的发展过程中可以发现 CSS 预处理器（例如 Less、SASS）在尽可能地使 CSS 像一门编程语言，但是始终没有解决 CSS 模块化的问题。

第一类 CSS 模块化的解决方案是彻底抛弃 CSS，使用 JavaScript 或 JSON 来写样式。这种做法能够给 CSS 提供 JavaScript 同样强大的模块化能力，但是使用 JavaScript 或 JSON 不能利用成熟的 CSS 预处理器 SASS/Less 等，同时:hover 和:active 伪类处理起来复杂。

另一类 CSS 模块化解决方案即 CSS Modules。CSS Modules 不是将 CSS 改造成编程语言，而是最大化地结合现有 CSS 生态和 JavaScript 模块化能力，依旧使用 CSS，但同时使用 JavaScript 来管理样式依赖，加入了局部作用域和模块依赖。因此，CSS Modules 很容易学，使用较少的规则实现网页组件最需要的功能，最重要的是可以保证某个组件的样式不会影响到其他组件。CSS Modules 是一种非常优秀的模块化解决方案。CSS Modules 发布时依旧编译出单独的 JavaScript 和 CSS。它并不依赖于 React，使用 webpack 可以在 Vue/Angular/jQuery 中使用。

8.2.2　局部样式

CSS 的规则默认都是全局的，任何一个组件的样式规则都对整个页面有效。CSS Modules 产生局部作用域的唯一方法就是使用一个独一无二的 class 的名字，不会与其他选择器重名。例如，编写一个 React 组件，如示例 8-2 所示。

【示例 8-2　React 组件】

```
import React from 'react';
import style from './App.css';

export default () => {
  return (
    <div className={style.wrapper}>
      <h1 className={style.title}>
        Hello World
      </h1>
      <h2 className={style.title}>
        CSS Modules
```

```
        </h2>
      </div>
    );
  };
```

在引入的 index.css 样式文件中编写样式表：

```
.wrapper {
  width: 750px;
  margin: 0 auto;
}
.title {
  color: red;
  font-size: 48px;
}
.sub-title {
  color: green;
  font-size: 36px;
}
```

在 webpack.config.js 中配置 css-loader 启用 CSS Modules：

```
module.exports = {
  entry:   dirname + '/index.js',
  output: {
    publicPath: '/',
    filename: './bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'stage-0', 'react']
        }
      },
      {
        test: /\.css$/,
        loader: "style-loader!css-loader?modules&localIdentName=
[name]__[local]-[hash:base64:5]"
      },
    ]
  }
```

```
};
```

加上 modules 即为启用，localIdentName 用于设置生成样式的命名规则。css-loader 默认的哈希算法是[hash:base64]，这会将.title 编译成._3zyde4l1yATCOkgn-DBWEL 这样的字符串。本示例中我们自定义的编译规则生成的 HTML 是：

```
<div data-reactroot=" class="App__wrapper-2N-zO">
  <h1 class="App__title-3zyde">Hello World</h1>
  <h2 class="App__subTitle-A_vr9">CSS Modules</h2>
</div>
```

App.css 也会同时被编译：

```
. App__wrapper-2N-zO {
  width: 750px;
  margin: 0 auto
}

. App__title-3zyde {
  color: red;
  font-size: 48px
}

. App__subTitle-A_vr9 {
  color: green;
  font-size: 36px
}
```

CSS Modules 对 CSS 中的 class 名都做了处理，使用对象来保存原 class 和混淆后 class 的对应关系：

```
t.locals={
  wrapper:" App__wrapper-2N-zO ",
  title:" App__title-3zyde ",
  "sub-title":" App__subTitle-A_vr9"
}
```

注意，App__wrapper-2N-zO、App__title-3zyde、App__subTitle-A_vr9 是 CSS Modules 按照 localIdentName 自动生成的 class 名。其中的 3zyde4l1yATCOkgn 是按照给定算法生成的序列码。经过这样混淆处理后，class 名基本就是唯一的，大大降低了项目中样式覆盖的概率。同时在生产环境下修改规则，生成更短的 class 名，可以提高 CSS 的压缩率。

通过这些处理，CSS Modules 可以继续使用 CSS 编写样式，相当于给每个 class 名外加了一个 :local，使得所有样式都是局部样式，以此来实现样式的局部化，解决了命名冲突和全局污染问题。

8.2.3 全局作用域

CSS Modules 还允许使用:global(.className)的语法，声明一个全局规则。凡是这样声明的 class，都不会被编译成哈希字符串。例如：

```
.normal {
  color: green;
}

/* 以上与下面等价 */
:local(.normal) {
  color: green;
}

/* 定义全局样式 */
:global(.btn) {
  color: red;
}

/* 定义多个全局样式 */
:global {
  .link {
    color: green;
  }
  .box {
    color: yellow;
  }
}
```

8.2.4 组合样式

在 CSS Modules 中，一个选择器可以继承另一个选择器的规则，称为“组合”。在 App.css 示例中，让.title 继承.titleBase，组件代码保持不变，如示例 8-3 所示。

【示例 8-3　组合样式】

```
import React from 'react';
import style from './App.css';

export default () => {
  return (
    <div className={style.wrapper}>
      <h1 className={style.title}>
        Hello World
      </h1>
```

```
        <h2 className={style.subTitle}>
          CSS Modules
        </h2>
      </div>
    );
  };
```

针对 App.css 样式代码，我们做如下修改：

```
  /* 通用样式 */
  .base {
    background-color: blue;
  }

  .wrapper {
    /* 组合 */
    composes: base;
    width: 750px;
    margin: 0 auto;
  }

  .title {
    /* 组合 */
    composes: base;
    /* 其他样式 */
    color: red;
    font-size: 48px;
  }

  .subTitle {
    color: green;
    font-size: 36px;
  }
```

运行编译之后，生成如下 HTML 代码：

```
  <div data-reactroot="" class="_34yD_SgZcBTCRJyysvxT7E
_22ntOO6juypA0UlgUkLjvV">
    <h1 class="_2DHwuiHWMnKTOYG45T0x34 _22ntOO6juypA0UlgUkLjvV">Hello
World</h1>
    <h2 class="O9H1hYQw4ds2_DiaeH9cz">CSS Modules</h2>
  </div>
```

App.css 编译成下面的代码：

```
  ._22ntOO6juypA0UlgUkLjvV {
```

```
    background-color: blue
  }

  ._34yD_SgZcBTCRJyysvxT7E {
    width: 750px;
    margin: 0 auto
  }

  ._2DHwuiHWMnKTOYG45T0x34 {
    color: red;
    font-size: 48px
  }

  .O9H1hYQw4ds2_DiaeH9cz {
    color: green;
    font-size: 36px
  }
```

原 class 与编译后的 class 对应关系如下：

```
   t.locals={
    base:"_22ntOO6juypA0UlgUkLjvV",
    wrapper:"_34yD_SgZcBTCRJyysvxT7E _22ntOO6juypA0UlgUkLjvV",
    title:"_2DHwuiHWMnKTOYG45T0x34 _22ntOO6juypA0UlgUkLjvV",
    subTitle:"O9H1hYQw4ds2_DiaeH9cz"
  }
```

由于在.title 中组合了.base，编译后 title 会变成两个 class。选择器也可以继承其他 CSS 文件里面的规则：

```
  /* settings.css */
  .primary-color {
    color: #f40;
  }

  /* components/App.css */
  .base { /* 通用的样式 */ }

  .title {
    composes: base;
    composes: primary-color from './settings.css';
    /*其他样式 */
  }
```

对于大多数项目，有了 composes 后已经不再需要 SASS/Less/PostCSS。由于 composes 不

是标准的 CSS 语法，因此编译时会报错，如果希望在项目中使用 CSS 预处理器，就只能使用预处理器自己的语法来做样式复用了。

CSS Modules 很好地解决了 CSS 目前面临的模块化难题，支持与 SASS/Less/PostCSS 等搭配使用，能充分利用现有技术积累。同时也能和全局样式灵活搭配，便于项目中逐步迁移至 CSS Modules。CSS Modules 的实现也属于轻量级，未来有标准解决方案后可以低成本迁移。

8.2.5　PostCSS

PostCSS 是一款对 CSS 进行处理的工具，主要依赖插件来进行操作。当我们需要某些功能的时候，只需配置相应的插件即可。PostCSS 有非常丰富的插件，可以涵盖开发过程的各个方面。即使没有满足项目需要的插件，开发者也可以使用 JavaScript 来开发自己的插件。PostCSS 是一个利用 JavaScript 插件来对 CSS 进行转换的工具，较常使用的插件有 CSS Modules、Autoprefixer、postcss-cssnext 等。

1. CSS Modules

在 CSS Modules 中支持使用变量，需要安装 PostCSS 和 postcss-modules-values：

```
npm install --save postcss-loader postcss-modules-values
```

【示例 8-4　CSS Modules】

在 webpack.config.js 中增加 postcss-loader 配置：

```
var values = require('postcss-modules-values');

module.exports = {
  entry: __dirname + '/index.js',
  output: {
    publicPath: '/',
    filename: './bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'stage-0', 'react']
        }
      },
      {
        test: /\.css$/,
        loader: "style-loader!css-loader?modules!postcss-loader"
```

```
      },
    ]
  },
  postcss: [
    values
  ]
};
```

在 colors.css 里面定义变量：

```
@value blue: #0c77f8;
@value red: #ff0000;
@value green: #aaf200;
```

在 App.css 中引用这些变量：

```
@value colors: "./colors.css";
@value blue, red, green from colors;

.title {
  color: red;
  background-color: blue;
}
```

2. Autoprefixer

针对浏览器兼容的处理可以使用 Autoprefixer 插件。Autoprefixer 是一个根据 Can I Use（http://caniuse.com）兼容性解析 CSS，然后为其添加浏览器厂商前缀的 PostCSS 插件。不加任何 vender prefix 的通常写法，例如：

```
::example{
display: none;
position:relative;
transform: translate(10, 10);
}
```

Autoprefixer 将使用基于当前浏览器支持的特性和属性数据来添加前缀，处理之后生成：

```
.example {
  display: none;
  position: relative;
  -webkit-transform: translate(10, 10);
      -ms-transform: translate(10, 10);
          transform: translate(10, 10);
}
```

display、position 属性没有浏览器差异，已经完全符合 W3C 标准的 CSS 2.1 属性，Autoprefixer 不会为其加前缀。针对 CSS3 属性，transform 会为其加前缀：--webkit 是 Chrome

和 Safari 前缀；--ms 则是 IE 浏览器的前缀；而 Firefox 由于已经实现了对 transform 的 W3C 标准化，因此直接使用 transform 即可。

3. postcss-cssnext

直接来看一个示例。

【示例 8-5　postcss-cssnext】

```
:root {
  --fontSize: 1rem;
  --mainColor: #12345678;
  --centered: {
      display: flex;
      align-items: center;
      justify-content: center;
  };
}
body {
    color: var(--mainColor);
    font-size: var(--fontSize);
    line-height: calc(var(--fontSize) * 1.5);
    padding: calc((var(--fontSize) / 2) + 1px);
}
.centered {
    @apply --centered;
}
```

通过 postcss-cssnext 处理之后：

```
body {
    color: rgba(18, 52, 86, 0.47059);
    font-size: 16px;
    font-size: 1rem;
    line-height: 24px;
    line-height: 1.5rem;
    padding: calc(0.5rem + 1px);
}
.centered {
    display: -webkit-box;
    display: -ms-flexbox;
    display: flex;
    -webkit-box-align: center;
        -ms-flex-align: center;
            align-items: center;
    -webkit-box-pack: center;
```

```
        -ms-flex-pack: center;
           justify-content: center;
  }
```

通过 var()和 calc()进行 CSS 属性值的计算，也有@apply 这样的应用大段规则的写法，可以借此去了解一些新的 CSS 草案特性。postcss-cssnext 正在尝试将 CSS 变为一种可以进行逻辑处理的语言。

8.3 状态管理

8.3.1 如何定义 state

定义一个合适的 state 是正确创建组件的第一步。state 必须能代表一个组件 UI 呈现的完整状态集，即组件的任何 UI 改变都可以从 state 的变化中反映出来；同时，state 还必须是代表一个组件 UI 呈现的最小状态集，即 state 中的所有状态都是用于反映组件 UI 变化的，没有任何多余的状态，也不需要通过其他状态计算而来的中间状态。

组件中用到的一个变量是不是应该作为组件 state，可以通过下面的 4 条依据进行判断：

- 这个变量是否是通过 Props 从父组件中获取？如果是，那么它不是一个状态。
- 这个变量是否在组件的整个生命周期中都保持不变？如果是，那么它不是一个状态。
- 这个变量是否可以通过其他状态（state）或者属性（Props）计算得到？如果是，那么它不是一个状态。
- 这个变量是否在组件的 render 方法中使用？如果不是，那么它不是一个状态。这种情况下，这个变量更适合定义为组件的一个普通属性，例如组件中用到的定时器，就应该直接定义为 this.timer，而不是 this.state.timer。

当然，并不是组件中用到的所有变量都是组件的状态！当存在多个组件共同依赖一个状态时，一般的做法是状态上移，将这个状态放到这几个组件的公共父组件中。

8.3.2 你可能不需要 Redux

在项目中使用 Redux 不是必需的，在以下几种情况下可能根本不需要 Redux：

- 项目中已经有一个预先定义的方式来共享和安排组件状态。
- 应用程序只包含大部分简单的操作（如 UI 更改），则这些操作并不一定是 Redux 存储的一部分，可以在组件级别进行处理。
- 不需要管理服务器端事件（SSE）、不需要与服务器大量交互，也没有使用 websockets。
- 可以从每个视图的单个数据源获取数据。

总之，可以借用“如果你不知道是否需要 Redux，那就是不需要它”这句话来概括。Redux

的创造者DanAbramov又补充了一句:"只有遇到React实在解决不了的问题,你才需要Redux。"从应用的角度考虑，Redux 的适用场景是：多交互、多数据源。从组件的角度考虑，Redux 适用于如下场景：

- 某个组件的状态需要共享。
- 某个状态需要在任何地方都可以拿到。
- 一个组件需要改变全局状态。
- 一个组件需要改变另一个组件的状态。

换句话说，Redux 不是必需的，Redux 只是 Web 架构中管理状态（变化和异步）的一种解决方案，也可以选择其他方案。

8.3.3　再来说说 Redux

随着 JavaScript 单页应用开发日趋复杂，JavaScript 在项目中需要管理更多 state（状态）。这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由、被选中的标签、是否显示加载动效或者分页器等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地可能会引起另一个 view 的变化。最终状态的追溯变得非常复杂，state 在什么时候、由于什么原因、如何变化已然不受控制。当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得异常复杂，甚至如何扩展新需求，如更新调优、服务端渲染、路由跳转前请求数据等都是前所未有的复杂性挑战。

React 只是 DOM 的一个抽象层，并不是 Web 应用的完整解决方案，有两个方面 React 没涉及：

- 代码结构。
- 组件之间的通信。

在 Redux 出现之前，在构建复杂任务时管理状态是相当痛苦的。受 Flux 应用程序设计模式的启发，Redux 设计用于管理 JavaScript 应用程序中的数据状态。虽然它主要用于 React，但是可以使用 Redux 与不同的框架和库（如 jQuery、Angular 或 Vue）。Redux 试图使用三个基本原则让 state 的变化变得可预测：

- 唯一数据源。
- 保持状态只读。
- 数据改变只能通过纯函数来完成。

Redux 确保应用程序的每个组件都可以直接访问应用程序的状态，而不必将 props 发送到子组件，或使用回调函数将数据发送回父组件。Redux 要求：

- 用简单的对象和数组来描述应用状态。

- 用简单对象来描述应用中的变更。
- 用纯函数来描述处理变更的逻辑。

Redux 中有 4 个核心概念，分别是 store、state、action、reducer，前面已经介绍过，这里简单复习一下：

- store 指的是存储数据的仓库，我们的数据只有放在这里才可以被 Redux 管理起来。Redux 提供了一个 createStore 来生成 store，其中 createStore 接收一个 reducers。
- state 指的是初始化的数据，这里的初始化数据是可以有多个的。
- action 指的是需要变化的数据，必须要有一个 type 参数和一些需要在 state 中修改的属性。
- reducer 因为 store 在收到我们传递过去的 action 之后需要对 state 进行更新，这个计算过程就叫作 reducer。reducer 是一个函数，接受 state 和 action 作为参数，返回一个新的 state。

Redux 提供的权衡是通过增加中间环节来将“发生了什么”和“该如何变化”进行解耦。例如，从组件中将 reducer 抽出：

```
import React, { Component } from 'react';

const counter = (state = { value: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { value: state.value + 1 };
    case 'DECREMENT':
      return { value: state.value - 1 };
    default:
      return state;
  }
}

class Counter extends Component {
  state = counter(undefined, {});

  dispatch(action) {
    this.setState(prevState => counter(prevState, action));
  }
  // 递增
  increment = () => {
    this.dispatch({ type: 'INCREMENT' });
  };
  // 递减
  decrement = () => {
    this.dispatch({ type: 'DECREMENT' });
```

```
    };

    render() {
      return (
        <div>
          {this.state.value}
          <button onClick={this.increment}>+</button>
          <button onClick={this.decrement}>-</button>
        </div>
      )
    }
  }
```



关于 Redux 的详细介绍请阅读第 7 章。

8.4 路由管理

本节介绍 React 体系的一个重要部分：路由管理 React-Router。React-Router 是官方维护的，也是唯一可选的路由库。React-Router 通过管理 URL 实现组件的切换和状态的变化，在开发复杂的应用时一定会用到。

React-Router 的作用是让 UI 组件与 URL 保持同步，在项目中可以通过简单的 API 实现强大的特性，例如代码懒加载、动态路由匹配、路径过渡处理等。

【示例 8-6　React-Router】

首先，安装 React-Router：

```
npm install react-router --save
```

Router 作为一个 React 组件，可以进行渲染：

```
import { Router, Route, hashHistory } from 'react-router';

render((
  <Router history={hashHistory}>
    <Route path="/" component={App}/>
  </Router>
), document.getElementById('app'));
```

Router 组件本身只是一个容器，真正的路由要通过 Route 组件定义。这里使用了 hashHistory 来管理路由历史与 URL 的哈希部分，路由的切换由 URL 的哈希变化决定，即 URL 的#部分发生变化。添加更多的路由，并指定它们对应的组件：

```
import About from './modules/About'
import Repos from './modules/Repos'

render((
  <Router history={hashHistory}>
    <Route path="/" component={App}/>
    <Route path="/repos" component={Repos}/>
    <Route path="/about" component={About}/>
  </Router>
), document.getElementById('app'))
```

用户访问/repos 时，会先加载 App 组件，然后在它的内部再加载 Repos 组件。App 组件代码示例：

```
export default React.createClass({
  render() {
    return <div>
      {this.props.children}
    </div>
  }
})
```

在上述示例代码中，App 组件的 this.props.children 属性就是子组件。当然，子路由也可以不写在 Router 组件里面，可以单独传入 Router 组件的 routes 属性，例如：

```
let routes = <Route path="/" component={App}>
  <Route path="/repos" component={Repos}/>
  <Route path="/about" component={About}/>
</Route>;

<Router routes={routes} history={browserHistory}/>
```

在上面的代码中，访问根路径/，不会加载任何子组件。也就是说，App 组件的 this.props.children 这时是 undefined。可以采用{this.props.children || <Home/>}这种写法，但是路由规则不清晰。因此可以改造成：

```
<Router>
  <Route path="/" component={App}>
    {* 根路由 *}
<IndexRoute component={Home}/>
{* 子路由 *}
    <Route path="accounts" component={Accounts}/>
    <Route path="statements" component={Statements}/>
  </Route>
</Router>
```

IndexRoute 显式指定 Home 是根路由的子组件，即指定默认情况下加载的子组件。现在，用户访问/的时候，加载的组件结构如下:

```
<App>
  <Home/>
</App>
```

App 只包含下级组件的共有元素，本身的展示内容则由 Home 组件定义。这样既有利于代码分离，也有利于使用 React Router 提供的各种 API。

我们想添加一个导航栏，保存在于每个页面上，如果没有路由器，就需要封装一个 nav 组件，并在每一个页面组件都引用和渲染。随着应用程序的增长，代码会显得很冗余。React Router 提供了一种方式来嵌套共享 UI 组件：

```
// index.js
// ...
render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      {/* 注意这里把两个子组件放在 Route 里，嵌套在了 App 的 Route 里/}
      <Route path="/repos" component={Repos}/>
      <Route path="/about" component={About}/>
    </Route>
  </Router>
), document.getElementById('app'))
```

接下来，在 App 中将子组件渲染出来：

```
// modules/App.js
// ...
  render() {
    return (
      <div>
        <h1>React Router Tutorial</h1>
        <ul role="nav">
          <li><Link to="/about">About</Link></li>
          <li><Link to="/repos">Repos</Link></li>
        </ul>
        {/* 注意这里将子组件渲染出来 */}
        {this.props.children}
      </div>
    )
  }
// ...
```

Link 组件可用于取代<a>元素，生成一个链接，允许用户单击后跳转到另一个路由。它基本上就是<a>元素的 React 版本，可以接收 Router 的状态：

```
// modules/App.js
import React from 'react'
import { Link } from 'react-router'

export default React.createClass({
  render() {
    return (
      <div>
        <h1>React Router Tutorial</h1>
        <ul role="nav">
          <li><Link to="/about">About</Link></li>
          <li><Link to="/repos">Repos</Link></li>
        </ul>
      </div>
    )
  }
})
```

上述示例使用了 Link 组件，可以渲染出链接并使用 to 属性指向相应的路由。

第 9 章 React服务端渲染

上一章介绍了 React 架构，本章开始介绍 React 服务端渲染（Server Side Render，SSR）。React 不仅能实现客户端渲染，还很好地支持服务端渲染。本章对服务端渲染的意义、原理以及实现方案展开讨论。

9.1 服务端渲染的意义

服务端渲染其实就是多页应用的原始做法，后台应用根据用户访问的不同 URL 路径，分别执行增、删、改、查等操作对应的模板到浏览器端，这在传统的 JSP、PHP 中就已经广泛使用。其中页面内容都是由后端模板生成，而前端 JavaScript 的作用更多的是做一些动态效果。

随着前后端分离越来越彻底，同时也得益于前端发展速度越来越快，前后端的合作模式逐渐演变成由后端提供 API（Application Programming Interface）接口、前端通过 AJAX 等异步获取数据的方法获取数据，而后由前端进行页面渲染。近些年来，随着 React/Vue 等框架的出现，前后端的这种开发方式越来越普及，基本成为标配，前端也越来越重要，网站也开始向单页面应用发展。

单页面应用即 SPA，是 Single Page Application 的缩写。然而，纯客户端渲染的单页面应用也带来一些棘手的问题，最常见的是单页应用不利于 SEO 的问题和首屏渲染性能的问题。

首先，纯客户端渲染的单页面应用的内容都是通过 JavaScript 完成渲染的（客户端渲染），即浏览器最初获取的是一个空的 HTML 文件，这就造成了 SEO 困难，搜索引擎几乎抓不到异步接口返回的内容，这种情况面向消费者的网站来说问题是非常严重的。

其次，纯客户端渲染的单页面应用的内容都是 JavaScript 生成的，而 JavaScript 异步获取到数据并进行渲染页面之前存在首页“白屏”问题（“白屏”是在完全由客户端呈现的 React 网站中可能发生的情况），这种情况相较于后端模板渲染完 HTML 再发送给浏览器的用户体验欠佳。

因此，现在讨论服务端渲染与传统的多页面网站服务器端渲染层次不同。我们所说的服务端渲染是在现有架构不变的情况下，即后端依旧只是提供 API 服务，前端人员依旧通过异步请求数据，同时要达到传统多页应用的首屏加载性能，且进行 SEO 优化到搜索引擎爬虫抓取工具可以直接查看完全渲染的页面（Google 有时会执行 JavaScript 程序并且对生成的内容进行索引，但并不总是的这样）。因为服务端渲染确实有着许多优势，尤其是 React 和 Node 相

结合实现前后端同构、前后端共用一套代码，更是将单页应用的便利和服务端渲染的好处相结合。这里来对比一下客户端渲染和服务端渲染的原理和过程，可参见图 9-1。

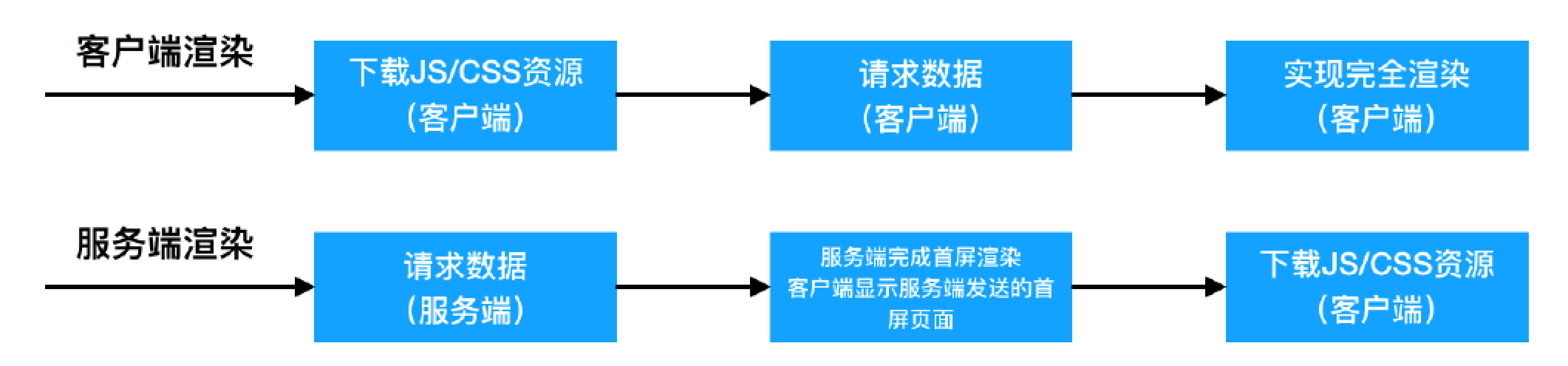


图 9-1　客户端渲染和服务端渲染对比

客户端渲染遵循如下过程：

（1）请求页面对应的 HTML。
（2）服务端返回对应的 HTML 文件。
（3）浏览器下载 HTML 文件中的 JavaScript/CSS 资源文件。
（4）等待 JavaScript/CSS 资源文件下载完成。
（5）等待 JavaScript/CSS 加载并初始化/渲染完成。
（6）运行 JavaScript 代码，由 JavaScript 代码向后端请求数据（ajax/fetch）。
（7）等待后端数据返回。
（8）客户端完成页面渲染。

相应地，服务端渲染遵循如下过程：

（1）请求页面对应的 HTML。
（2）服务端请求数据（内网请求快）。
（3）服务器初始渲染（服务端性能优秀）。
（4）服务端返回已经有正确内容的页面。
（5）客户端请求 JavaScript/CSS 资源文件。
（6）等待 JavaScript/CSS 资源文件下载完成。
（7）等待 JavaScript/CSS 加载并初始化/渲染完成。
（8）客户端把剩下的一部分渲染完成（可懒加载）。

无论是客户端渲染还是服务端渲染，都包含三个主体过程：下载 JavaScript/CSS 文件、请求数据、渲染页面。其中，客户端渲染执行的顺序是“下载 JavaScript/CSS 文件”→“请求数据”→“渲染页面”，三个过程都在客户端进行；服务端渲染执行的顺序是“请求数据”→“渲染页面”→“下载 JavaScript/CSS 文件”，其中请求数据和渲染页面在服务端进行，最后下载 JavaScript/CSS 文件在客户端进行。

服务端渲染改变了三个过程的执行顺序和执行服务器，返回到客户端的是已经有正确内容的页面，可直接用于 SEO。同时，相比于客户端首屏渲染，服务端首屏渲染不需要在客户端下载 JavaScript/CSS 文件，客户端接收服务端内容的时候，接收的已经是完整的可视页面。服

务端在内网请求数据（拉取数据），数据响应速度是很快的；而对于客户端渲染，外网 HTTP 请求开销大，且受到具体的网络环境的限制。因此，服务端渲染的首屏渲染比客户端渲染性能更优。

9.2 理解服务端渲染原理

上一节描述了客户端渲染和服务端渲染，实际上分别对应了两种 Web 构建模式：前后分离模式和直出模式。前后分离模式对应客户端渲染（见图 9-2），直出模式对应服务端渲染（见图 9-3）。

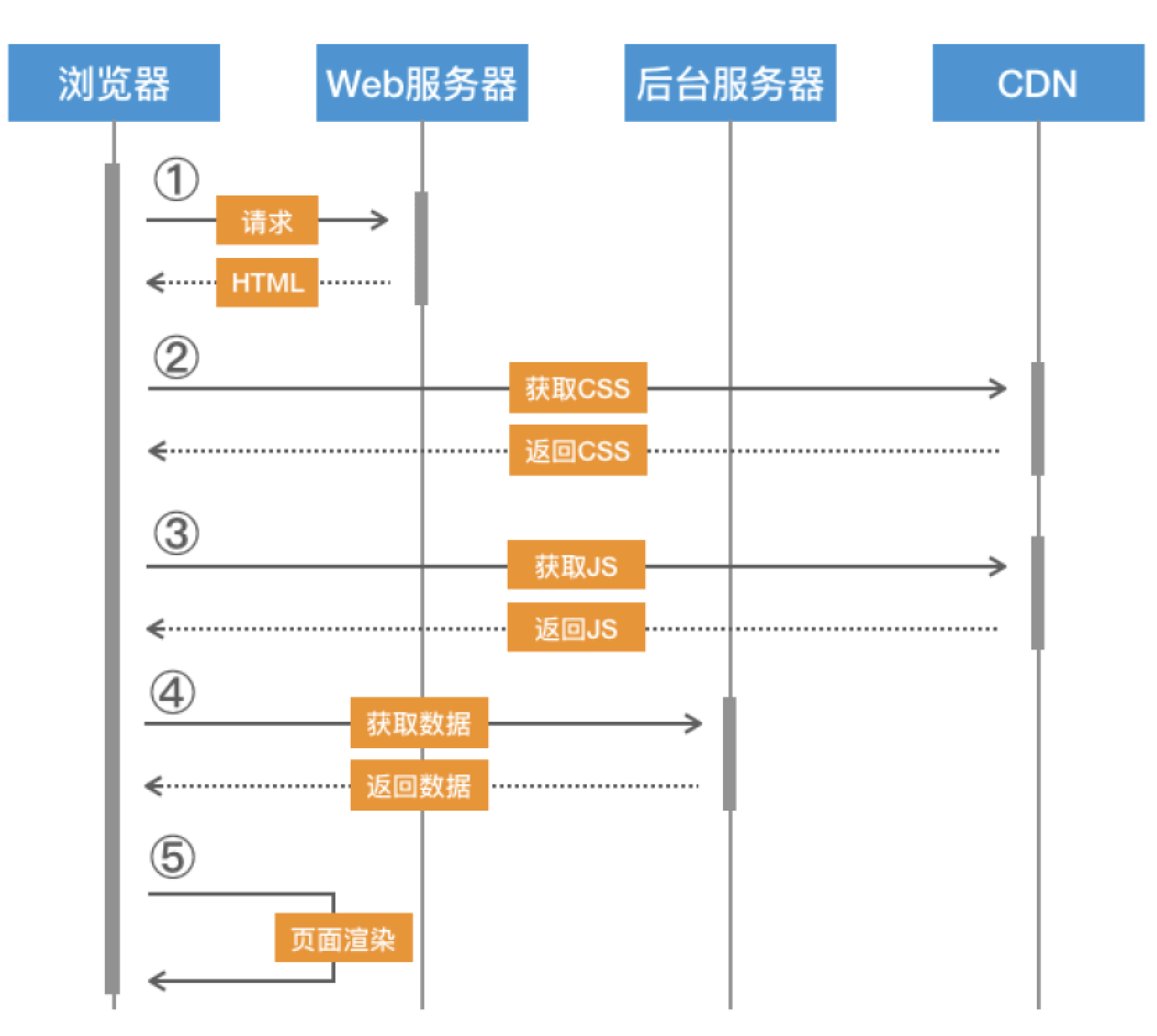


图 9-2　客户端渲染

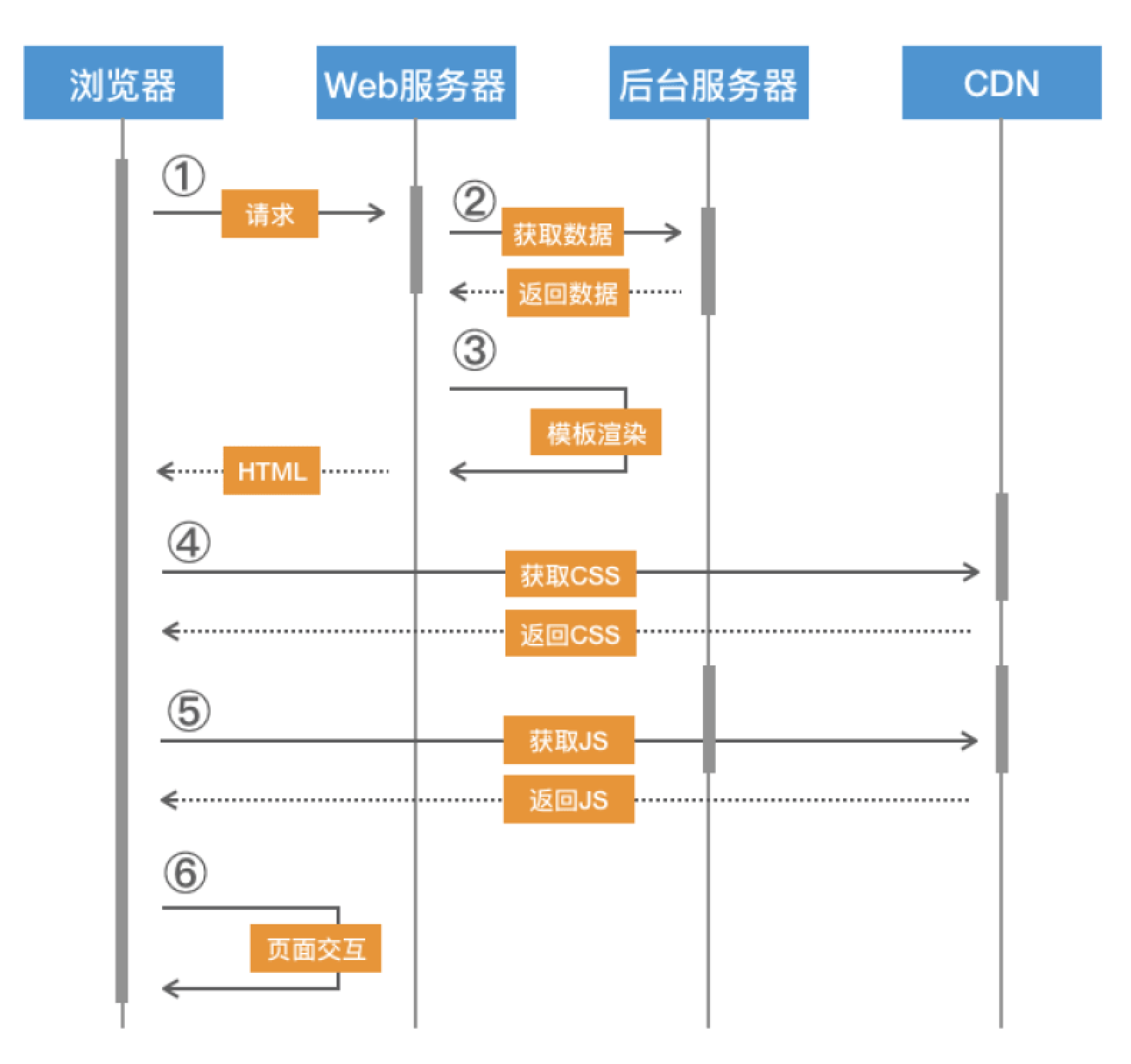


图 9-3　服务端渲染

相对于客户端渲染来说，服务端渲染的核心保障是首屏渲染。服务端渲染需要在请求 HTML 时直接返回渲染好的首屏页面（包括所需的服务端数据）。使用 Node 和 React 相结合的前后端同构、前后端共用一套代码，对应的服务端渲染的原理如图 9-4 所示。

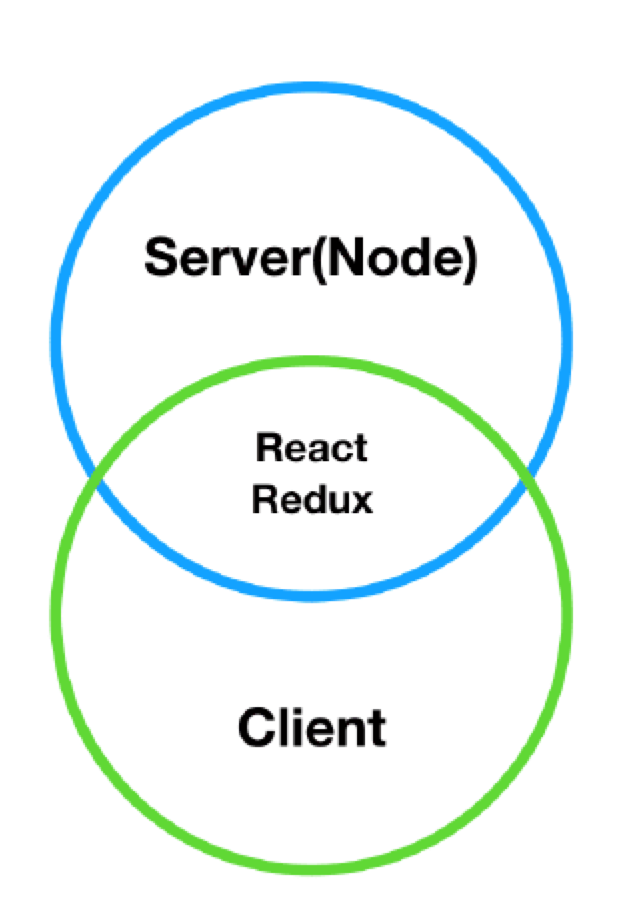


图 9-4　服务端渲染原理

从图 9-4 可以看出，我们希望尽可能复用同一份代码实现客户端渲染和服务端渲染，以便于开发和后期维护。在前后端渲染相同的 Component，将输出一致的 DOM 结构。完善的 Component 属性及生命周期与客户端的 render 时机是 React 同构的关键。React 的虚拟 DOM 以对象树的形式保存在内存中，并且是可以在任何支持 JavaScript 的环境中生成的，所以可以在浏览器和 Node 中生成，这为前后端同构提供了先决条件。

如图 9-5 所示，React 的虚拟 DOM 是可以在任何支持 JavaScript 的环境中生成的，所以可以在浏览器 Node 环境中生成。虚拟 DOM 既可以直接转成 String，也可以直接输入到 HTML 文件中返回给浏览器。

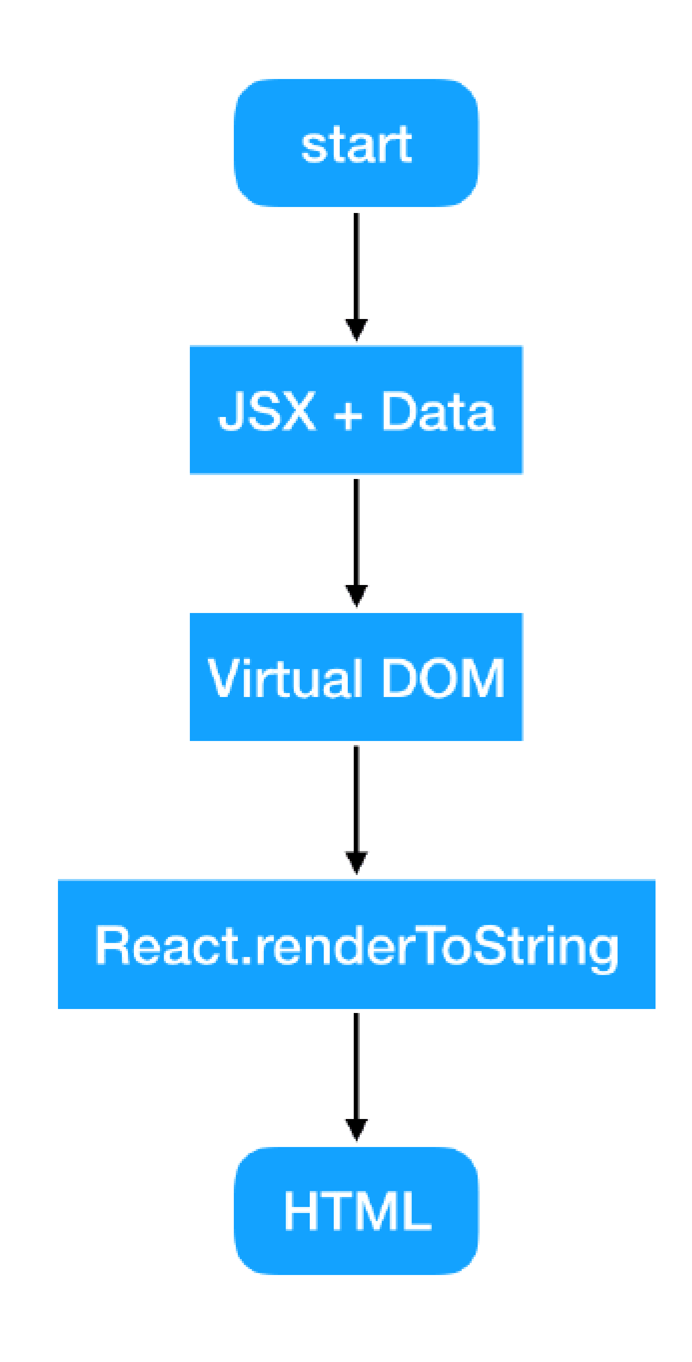


图 9-5　服务端生成 HTML

虚拟 DOM 在前后端都是以对象树的形式存在的，但展露原型的方式却有所不同，如图 9-6 所示。①在浏览器里，React 通过 ReactDom 的 render 方法将虚拟 DOM 渲染到真实的 DOM 树上，生成网页。②在 Node 环境下是没有渲染引擎的，所以 React 提供了另外两个方法，即 ReactDOMServer.renderToString 和 ReactDOMServer.renderToStaticMarkup，可将其渲染为 HTML 字符串。

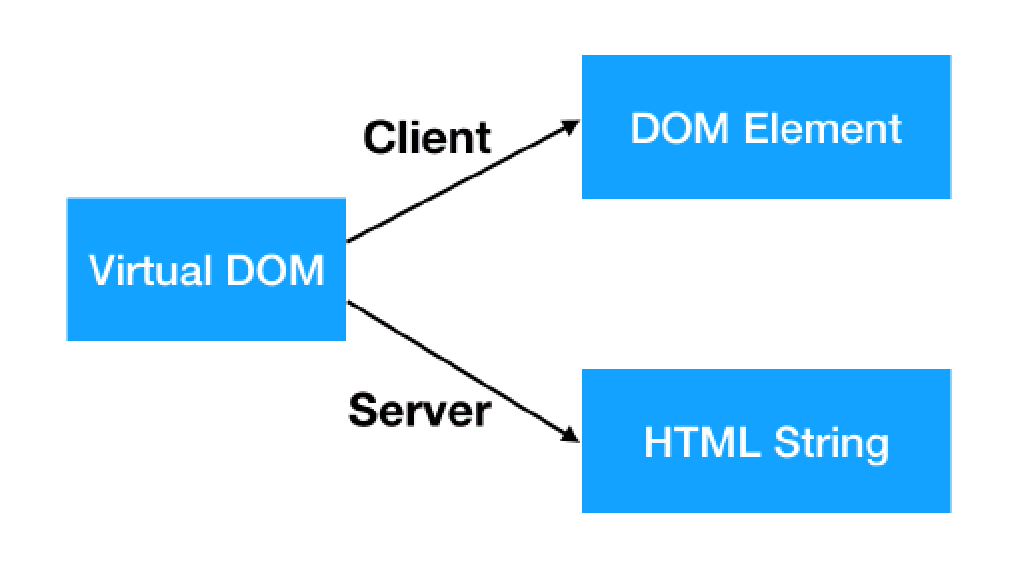


图 9-6　虚拟 DOM

服务端结合数据将 Component 渲染成完整的 HTML 字符串并将数据状态返回给客户端，客户端会判断是否可以直接使用或需要重新挂载。这些是 React 在服务端渲染提供的基础条件。在实际项目应用中，还需要考虑其他问题。例如，服务器端没有 window 对象，需要做不同处理等。

9.3 实战：动手实现服务端渲染

了解了服务端渲染的意义和基本渲染原理之后，本节开始动手实现服务端渲染。

（1）首先准备一个简单的 React 项目，例如：

```
import React from 'react';
import ReactDOM from 'react-dom';
// 引入 redux
import {
    createStore,
    applyMiddleware,
    compose
} from "redux";
import { Provider } from "react-redux";
// 引入 thunk 中间件
import thunk from "redux-thunk";
// 引入组件
import {
    BrowserRouter
} from "react-router-dom";
// 引入 reducer
import reducers from "./reducer";
// 引入路由
import Routers from './router'
// 引入 antd css
import 'antd-mobile/dist/antd-mobile.css';
// 创建 store
const store = createStore(reducers,compose(
    applyMiddleware(thunk),
));
// 渲染组件
ReactDOM.render(
    <Provider store={store}>
        <BrowserRouter>
            <Routers/>
        </BrowserRouter>
    </Provider>,
    document.getElementById('root')
);
```

在构建完成的项目目录中生成 build 目录，build 目录中含有一个 asset-manifest.json 的文件，该文件存储的是打包后 JavaScript、CSS 文件的路径，可以看到每个路径后面都有一个哈希值，

每次打包生成的哈希值都不同，可用于区分版本，为以后做服务端渲染做好铺垫。

（2）接着，我们在本地搭建一个 server。修改 server 文件的 server.js，代码如下：

```
// 引入 express
const express = require("express");
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
const userRoute = require("./userRoute");
// 创建实例
const app = express();
const path = require('path');
app.use(cookieParser());
app.use(bodyParser.json());

// 用户接口模块
app.use("/user",userRoute);

// 映射到 build 后的路径
// 设置 build 以后的文件路径 项目上线用
app.use((req, res, next) => {
    if (req.url.startsWith('/user/') || req.url.startsWith('/static/')) {
        return next()
    }
    return res.sendFile(path.resolve('build/index.html'))
})
// 路由
app.use('/', express.static(path.resolve('build')))
// 设置端口
app.listen("9000",function(){
    console.log("open Browser http://localhost:9000");
});
```

（3）我们在 package.json 中增加一条命令，scripts 如下：

```
"scripts": {
    "start": "node scripts/start.js",
    "build": "node scripts/build.js",
    "test": "node scripts/test.js --env=jsdom",
    "server": "nodemon server/server.js"
},
```

（4）运行 npm run server，在浏览器中打开 http://localhost:9000/，可以看到项目已经运行成功，此时我们的端口是 9000，并且代码运行的都是 build 以后的代码。

（5）React.createElement 把 React 类进行实例化，实例化后的组件就可以进行 mount 操

作， 在浏览器环境中我们是使用 ReactDOM.render() 来进行渲染操作的。ReactDOMServer.renderToString 则是把 React 实例渲染成 HTML 标签。接下来需要把 index.js 中的代码挪到 server.js 中，渲染成 HTML 返回给客户端即可。进行代码改造，完成后如下：

```
// 引入 express
const express = require("express");
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
const userRoute = require("./userRoute");
// 创建实例
const app = express();
const path = require('path');
app.use(cookieParser());
app.use(bodyParser.json());

/**
 * 插入 react 代码 进行服务端改造
 */
import React from 'react';
import ReactDOM from 'react-dom';
// 引入 redux
import {
    createStore,
    applyMiddleware,
    compose
} from "redux";
import { Provider } from "react-redux";
import thunk from "redux-thunk";
// 引入 renderToString
import { renderToString, renderToStaticMarkup } from 'react-dom/server';
// 服务端没有 BrowserRouter，所以用 StaticRouter
import { StaticRouter } from "react-router-dom";
// 引入 reducer
import reducers from "../src/reducer";
// 引入前端路由
import Routers from '../src/router'

const store = createStore(reducers, compose(
    applyMiddleware(thunk),
));

// 用户接口模块
app.use("/user", userRoute);
```

```
// 映射到 build 后的路径
//设置 build 以后的文件路径 项目上线用
app.use((req, res, next) => {
    if (req.url.startsWith('/user/') || req.url.startsWith('/static/')) {
        return next()
    }
    const context = {}
    const frontComponents = renderToString(
        (<Provider store={store}>
            <StaticRouter
                location={req.url}
                context={context}>
                <Routers />
            </StaticRouter>
        </Provider>)
    )
    res.send(frontComponents)
    // return res.sendFile(path.resolve('build/index.html'))
})
// 服务端路由
app.use('/', express.static(path.resolve('build')))
// 设置端口
app.listen("9000", function () {
    console.log("open Browser http://localhost:9000");
});
```

可以看到我们将前端代码用 renderToString()处理后返回给前端，这样前端就能接收到首屏加载所需要的代码了，但是一个完整的页面需要由 HTML 等标签元素构成的，还缺少一个支持页面的骨架，需要在服务端加上返回给前端。

（6）对 server.js 进行改造，代码如下：

```
// 处理 css
import csshook from 'css-modules-require-hook/preset';
// 处理图片
import assethook from 'asset-require-hook';
assethook({
    extensions: ['png', 'jpg']
});

const express = require("express");
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
```

```
const userRoute = require("./userRoute");
const app = express();
const path = require('path');
app.use(cookieParser());
app.use(bodyParser.json());

/**
 * 插入 react 代码 进行服务端改造
 */
import React from 'react';
import ReactDOM from 'react-dom';
// 引入 redux
import {
    createStore,
    applyMiddleware,
    compose
} from "redux";
import { Provider } from "react-redux";
// 引入 thunk 中间件
import thunk from "redux-thunk";
// 引入 antd css
import 'antd-mobile/dist/antd-mobile.css';
// 引入 renderToString
import { renderToString, renderToStaticMarkup } from 'react-dom/server';
// 服务端没有 BrowserRouter，所以用 StaticRouter
import { StaticRouter } from "react-router-dom";
// 引入 reducer
import reducers from "../src/reducer";
// 引入前端路由
import Routers from '../src/router'
// 创建 store
const store = createStore(reducers, compose(
    applyMiddleware(thunk),
));

// 用户接口模块
app.use("/user", userRoute);

// 映射到 build 后的路径
//设置 build 以后的文件路径 项目上线用
app.use((req, res, next) => {
    if (req.url.startsWith('/user/') || req.url.startsWith('/static/')) {
        return next()
```

```
        }
        const context = {}
        const frontComponents = renderToString(
            (<Provider store={store}>
                <StaticRouter
                    location={req.url}
                    context={context}>
                    <Routers />
                </StaticRouter>
            </Provider>)
        )
        // 新建骨架
        const _frontHtml = `<!DOCTYPE html>
        <html lang="en">
            <head>
                <meta charset="utf-8">
                <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
                <meta name="theme-color" content="#000000">
                <title>人才市场</title>
            </head>
            <body>
                <noscript>
                You need to enable JavaScript to run this app.
                </noscript>
                <div id="root">${frontComponents}</div>
            </body>
        </html>`
        res.send(_frontHtml)
        // return res.sendFile(path.resolve('build/index.html'))
    })
    // 路由
    app.use('/', express.static(path.resolve('build')))
    // 端口
    app.listen("9000", function () {
        console.log("open Browser http://localhost:9000");
    });
```

（7）此时刷新页面后，打开 Network 下面的 response，可以看到服务端已返回结果：

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no" />
    <meta name="theme-color" content="#000000" />
    <title>人才市场</title>
   </head>
   <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root">
     <div data-reactroot="">
      <div class="bigbox">
       <div class="logo-container">
        <img src="72ba71de2dfbe02c990266c62394b476.png" alt="" />
       </div>
       <h1 style="color:red;text-align:center">React SSR </h1>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-whitespace am-whitespace-md"></div>
       <div class="am-wingblank am-wingblank-lg">
        <div class="am-list">
         <div class="am-list-header"></div>
         <div class="am-list-body">
          <div class="am-list-item am-input-item am-list-item-middle">
           <div class="am-list-line">
            <div class="am-input-label am-input-label-5">
             <i class="iconfont icon-yonghu c-blue"></i>
            </div>
            <div class="am-input-control">
             <input type="text" placeholder="请输入用户名" value="" />
            </div>
           </div>
          </div>
          <div class="am-whitespace am-whitespace-md"></div>
```

```
            <div class="am-whitespace am-whitespace-md"></div>
            <div class="am-list-item am-input-item am-list-item-middle">
             <div class="am-list-line">
              <div class="am-input-label am-input-label-5">
               <i class="iconfont icon-mima c-blue" style="font-size:19px"></i>
              </div>
              <div class="am-input-control">
               <input type="password" placeholder="请输入密码" value="" />
              </div>
             </div>
            </div>
           </div>
          </div>
          <div class="am-whitespace am-whitespace-md"></div>
          <div class="ta-right">
           <a href="/">忘记密码？</a>
          </div>
          <div class="am-whitespace am-whitespace-md"></div>
          <div class="am-whitespace am-whitespace-md"></div>
          <a role="button" class="am-button am-button-primary"
aria-disabled="false"><span>登 录</span></a>
          <div class="am-whitespace am-whitespace-md"></div>
          <a role="button" class="am-button am-button-primary"
aria-disabled="false"><span>注 册</span></a>
          <div class="am-whitespace am-whitespace-md"></div>
          <div class="am-whitespace am-whitespace-md"></div>
         </div>
        </div>
       </div>
      </div>
     </body>
    </html>
```

（8）该 HTML 中没有引入 CSS 和 JavaScript 文件，build 目录中 asset-manifest.json 的文件中有所需的 CSS 和 JavaScript 文件，我们引入它，就可以拿到每次 build 以后最新的代码，对 server.js 进行代码改造：

```
// 处理 css
import csshook from 'css-modules-require-hook/preset';
// 处理图片
import assethook from 'asset-require-hook';
assethook({
   extensions: ['png', 'jpg']
});
```

```
const express = require("express");
const bodyParser = require("body-parser");
const cookieParser = require("cookie-parser");
const userRoute = require("./userRoute");
const app = express();
const path = require('path');
app.use(cookieParser());
app.use(bodyParser.json());

/**
 * 插入 react 代码 进行服务端改造
 */
import React from 'react';
import ReactDOM from 'react-dom';
import {
    createStore,
    applyMiddleware,
    compose
} from "redux";
import { Provider } from "react-redux";
import thunk from "redux-thunk";
// 引入 antd css
import 'antd-mobile/dist/antd-mobile.css';
// 引入 renderToString
import { renderToString, renderToStaticMarkup } from 'react-dom/server';
// 服务端没有 BrowserRouter，所以用 StaticRouter
import { StaticRouter } from "react-router-dom";
// 引入 reducer
import reducers from "../src/reducer";
// 引入前端路由
import Routers from '../src/router';
// 引入 css 和 js
import buildPath from '../build/asset-manifest.json';

const store = createStore(reducers, compose(
    applyMiddleware(thunk),
));

// 用户接口模块
app.use("/user", userRoute);

// 映射到 build 后的路径
```

```
    //设置 build 以后的文件路径 项目上线用
    app.use((req, res, next) => {
        if (req.url.startsWith('/user/') || req.url.startsWith('/static/')) {
            return next()
        }
        const context = {}
        const frontComponents = renderToString(
            (<Provider store={store}>
                <StaticRouter
                    location={req.url}
                    context={context}>
                    <Routers />
                </StaticRouter>
            </Provider>)
        )
        // 新建骨架
        const _frontHtml = `<!DOCTYPE html>
        <html lang="en">
            <head>
                <meta charset="utf-8">
                <meta name="viewport" content="width=device-width, initial-scale=1, 
shrink-to-fit=no">
                <meta name="theme-color" content="#000000">
                <title>人才市场</title>
                <link rel="stylesheet" type="text/css" 
href="/${buildPath['main.css']}">
            </head>
            <body>
                <noscript>
                You need to enable JavaScript to run this app.
                </noscript>
                <div id="root">${frontComponents}</div>
                <script src="/${buildPath['main.js']}"></script>
            </body>
        </html>`
        res.send(_frontHtml)
        // return res.sendFile(path.resolve('build/index.html'))
    })
    app.use('/', express.static(path.resolve('build')))

    app.listen("9000", function () {
        console.log("open Browser http://localhost:9000");
    });
```

（9）刷新页面后可以看到基本是我们想要的页面了，后台已经返回一个完整的 HTML：

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no" />
  <meta name="theme-color" content="#000000" />
  <title>人才市场</title>
  <link rel="stylesheet" type="text/css" href="/static/css/main.f4dbdb58.css" />
 </head>
 <body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root">
   <div data-reactroot="">
    <div>
     <div class="am-navbar am-navbar-dark">
      <div class="am-navbar-left" role="button"></div>
      <div class="am-navbar-title">
       消息列表
      </div>
      <div class="am-navbar-right"></div>
     </div>
     <div class="mt-45 mb-50">
      <div>
       msgpages
      </div>
     </div>
     <div class="am-tab-bar">
      <div class="am-tabs am-tabs-horizontal am-tabs-bottom">
       <div class="am-tabs-content-wrap" style="touch-action:pan-x
pan-y;position:relative;left:-200%">
        <div class="am-tabs-pane-wrap am-tabs-pane-wrap-inactive"></div>
        <div class="am-tabs-pane-wrap am-tabs-pane-wrap-inactive">
         <div class="am-tab-bar-item"></div>
        </div>
        <div class="am-tabs-pane-wrap am-tabs-pane-wrap-active">
         <div class="am-tab-bar-item"></div>
        </div>
        <div class="am-tabs-pane-wrap am-tabs-pane-wrap-inactive">
         <div class="am-tab-bar-item"></div>
```

```
            </div>
           </div>
           <div class="am-tabs-tab-bar-wrap">
            <div class="am-tab-bar-bar" style="background-color:white">
             <div class="am-tab-bar-tab">
              <div class="am-tab-bar-tab-icon" style="color:#888">
               <img class="am-tab-bar-tab-image"
src="a12c878ee5f7d318376da191b5b76ef7.png" alt="BOSS" />
              </div>
              <p class="am-tab-bar-tab-title" style="color:#888">BOSS</p>
             </div>
             <div class="am-tab-bar-tab">
              <div class="am-tab-bar-tab-icon" style="color:#888">
               <img class="am-tab-bar-tab-image"
src="c6c95d08bf1b9a888cce1053bfa2bf18.png" alt="牛人" />
              </div>
              <p class="am-tab-bar-tab-title" style="color:#888">牛人</p>
             </div>
             <div class="am-tab-bar-tab">
              <div class="am-tab-bar-tab-icon" style="color:#108ee9">
               <img class="am-tab-bar-tab-image"
src="f73fc85762cfbe7999c792ce031c7fce.png" alt="消息" />
              </div>
              <p class="am-tab-bar-tab-title" style="color:#108ee9">消息</p>
             </div>
             <div class="am-tab-bar-tab">
              <div class="am-tab-bar-tab-icon" style="color:#888">
               <img class="am-tab-bar-tab-image"
src="3bf7c1d72788277ba13047821af2d180.png" alt="我的" />
              </div>
              <p class="am-tab-bar-tab-title" style="color:#888">我的</p>
             </div>
            </div>
           </div>
          </div>
         </div>
        </div>
       </div>
      </div>
      <script src="/static/js/main.7993f0a1.js"></script>
     </body>
    </html>
```

一个简单的、基于 React 的服务端渲染已完成。

9.4 服务器渲染的思考

本章介绍了服务端渲染的意义、渲染原理和渲染示例。值得思考的是，随着用户个性化算法的深入，服务端渲染需要面对个性化缓存的问题。试想把每个用户个性化信息全部缓存放到服务器存储，需要的存储空间和计算都是非常大的，这部分信息若由用户浏览器存储则能形成类似分布式存储的效果。同时，在做前后端代码复用时需要慎重，前端代码在编写时需仔细考虑后端渲染的情景，慎用 BOM 对象和 DOM API。

做服务端渲染同构之前，一定要考虑到浏览器和服务器的环境差异，站在更高层面考虑。Next.js 是时下非常流行的基于 React 的同构开发框架，提供了异步请求、样式、拆分文件打包的整体解决方案。

第 10 章 编写测试

测试是一种比较实际输出与预期输出之间差异的过程，通过测试可以衡量代码的质量和评估能否满足实际需求。所有的代码均需经过测试才允许发布上线。本章介绍测试驱动开发的好处、现状，以及如何使用 React 测试工具。

10.1 测试驱动开发

测试驱动开发（Test-Driven Development，TDD）是敏捷开发中的一项核心实践和技术，也是一种设计方法论，其基本思想是在明确所需开发的功能之后，在着手编写功能代码之前，优先进行测试代码的编写，随后编写功能代码并使用测试代码进行验证，如此循环直到完成全部的功能开发。

测试驱动开发有广义和狭义之分，常说的是狭义的测试驱动开发，也就是单页测试驱动开发（Unit Test Driven Development，UTDD）。广义的 TDD 是验收测试驱动开发（Acceptance Test Driven Development，ATDD），包括行为驱动开发（Behavior Driven Development，BDD）和以服务消费者定义契约为驱动的开发模式（Consumer-Driven Contracts Development） 等。我们这里说的 TDD（测试驱动开发），实际上可以更准确地描述为单元测试驱动开发。

测试驱动开发的核心目标是编写出优秀、高质量、具有可维护性的和可扩展性的代码。

10.1.1 测试驱动开发的好处

或许有许多开发者不喜欢写测试用例代码，觉得编写测试用例是在浪费时间，而且测试用例代码维护起来也是非常烦琐。但是当你尝试开始写测试代码的时候，特别是基础组件类的，就会发现测试代码的好处，不但能提高组件的代码质量，而且当依赖库发生更新、版本变化时，能够立刻发现这些潜在的问题和风险。如果没有测试代码，就更谈不上自动化测试了。

测试驱动开发有以下几项好处。

（1）保证代码质量

通过明确的流程，让开发者一次只关注一个功能点，分离关注点，一次只做一件重要的事、只关注一个重要的方面，减小思维负担，注重代码质量，编写结构清晰的代码。

（2）保证代码与业务需求的一致性

提前编写测试用例可以帮助我们去思考需求细节和边界，对需求范围进行细致梳理，避免代码编写过程中才发现需求信息不对称的情况。测试驱动开发能够完全覆盖所有的单元测试，对产品代码提供了一个保护措施。

（3）自动化测试、回归测试，确保新的更改不影响现有功能

测试驱动开发能够在保证项目的代码与所需的业务匹配的同时，让开发者轻松地接受新的需求变化或改善代码的设计，并进行自动化测试和回归测试，确保新的更改不影响现有功能，保证之前功能的正确与完整性，减少不必要的问题。

（4）提升测试效率

从一个角度看，编写测试用例会造成所需编写的代码量增加，造成开发效率的降低。但是，如果没有单元测试，我们就需要手动测试，甚至仍然需要花费许多时间和精力准备测试数据，完整的测试下来反馈链路非常长。准确地说，快速反馈是单元测试的一大优势。

（5）提升系统的开放性和扩展性

为了实现测试驱动开发，提前思考程序设计模式与解耦，有利于提升系统的开放性和扩展性，便于协同开发和多功能模块的组装。

10.1.2　测试驱动开发现状

传统开发模式流程通常是接手需求之后立刻进行项目代码开发，开发完成之后再进行测试用例编写，然后运行测试用例，发现并修复代码缺陷，如图 10-1 所示。

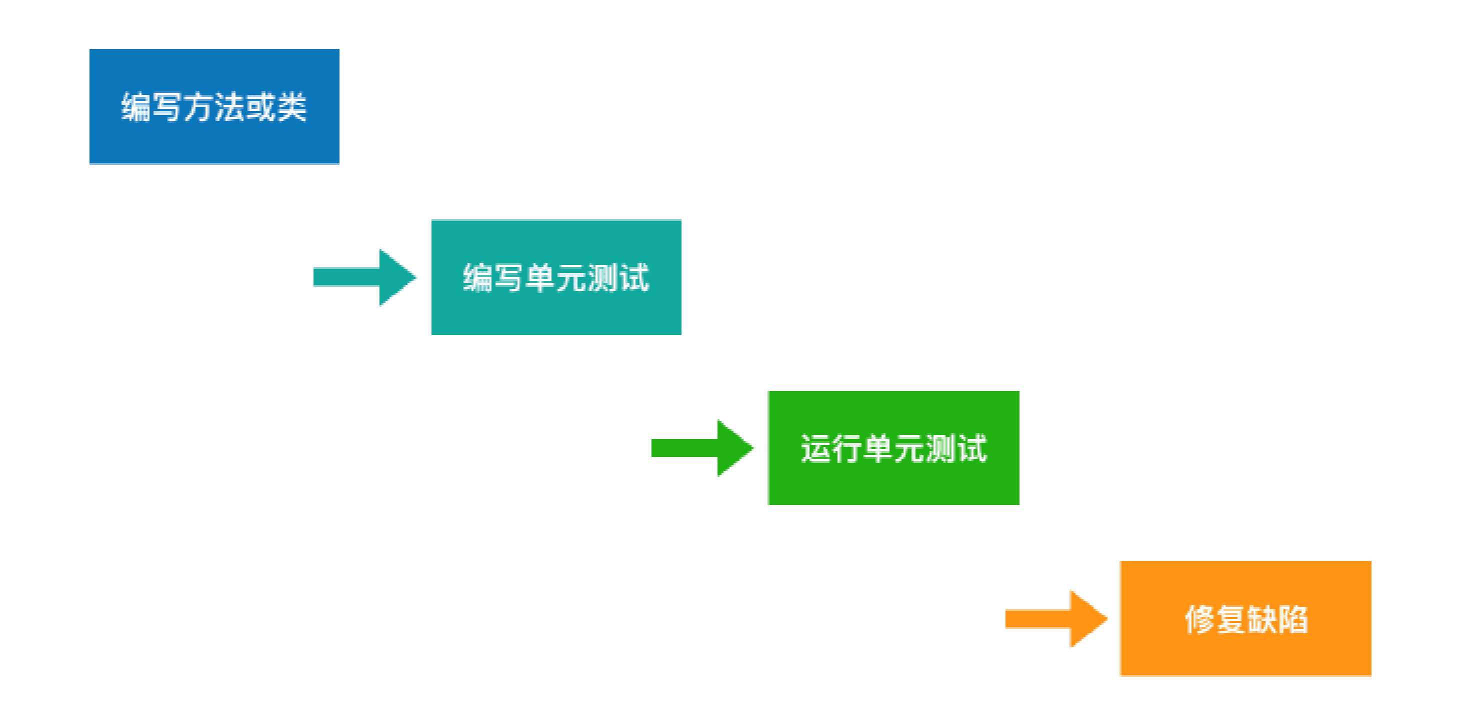


图 10-1　传统开发流程

测试驱动开发模式流程则是优先编写测试用例，运行测试用例与编写项目代码交替进行，最后重构/组装代码，如图 10-2 所示。

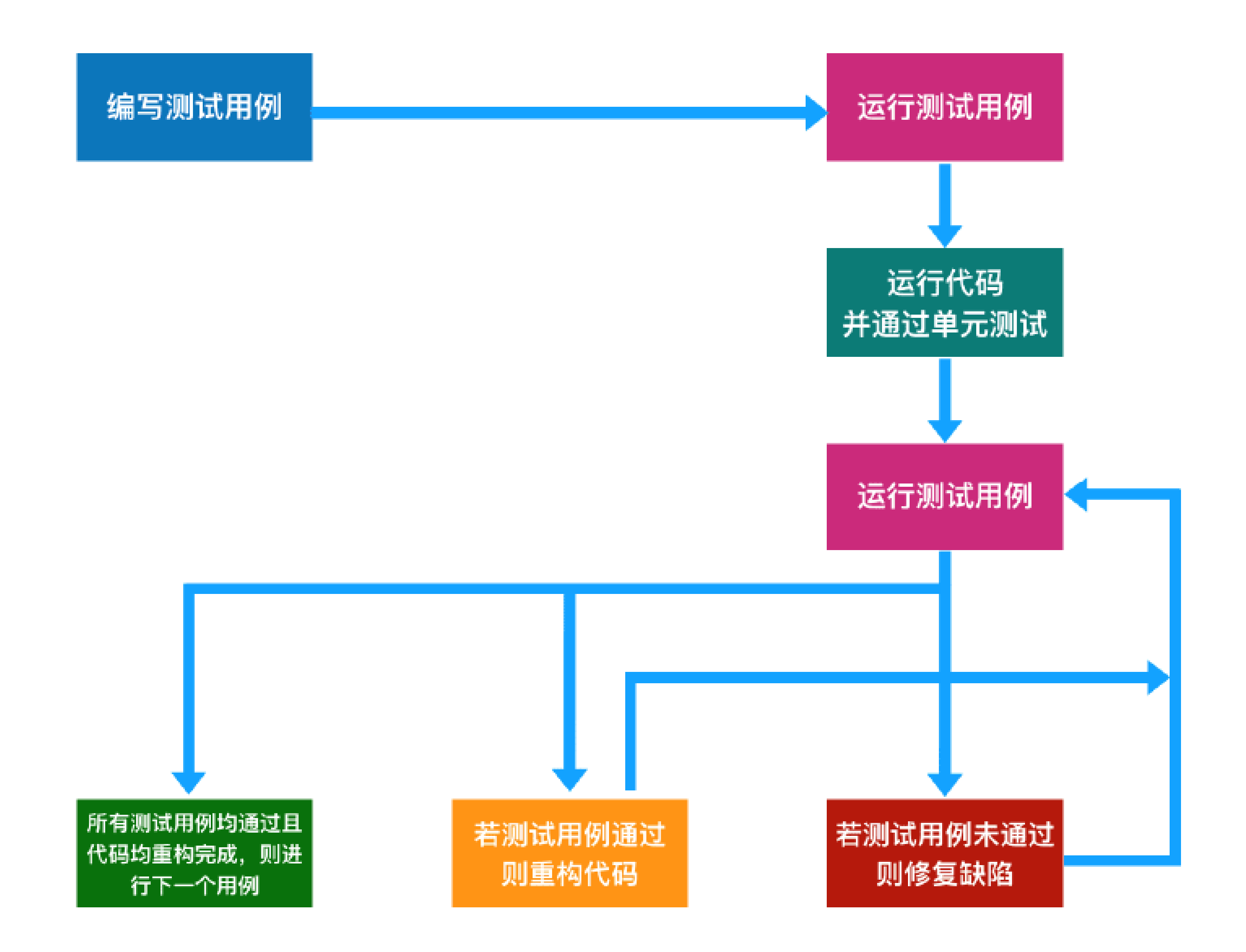


图 10-2　测试驱动开发流程

测试驱动开发已得到越来越广泛的重视，如今越来越多的公司都在尝试实践测试驱动开发。由于测试驱动开发对开发人员要求比较高，需要开发人员的转变思维习惯，逐步适应并提升效率。

10.1.3　定义属于自己的测试原则

（1）测试驱动

测试驱动开发的基本思想是在开发功能代码前先开发测试代码，并用测试代码验证功能实现是否满足需求或存在缺陷，在测试代码的驱动下优化功能代码的开发。

（2）独立测试

测试代码的作用是在被测代码发生改动后，执行单元测试用例即可验证本次改动是否对函数原有功能造成影响，是未来函数重构的信心保证。测试驱动开发的实施手段是单元测试，在每次版本改动后，使用测试用例验证了版本修复情况，同时也验证了本次改动是否引起回归问题。

（3）可测试性

在产品代码设计、开发时应尽可能提高可测试性。每个代码单元的功能应该尽可能纯粹简明，每个类、每个函数应该只做它该做的事，避免耦合。尤其是增加新功能时，不要为了图一时之便，随便在原有代码中添加功能。

（4）及时重构

结构不合理、重复等不优秀的代码，在测试通过后，应及时进行重构。

10.2 React 测试工具

前端有非常多的工具可以选择，比如 Mocha、Jasmine、Karma、Jest 等，要从这些工具里面选择一个也是比较困难的问题。React 的组件结构和 JSX 语法不适用传统的测试工具，必须有新的测试方法和工具，现在主流推荐使用 Jest 作为测试框架、Enzyme 作为 React 组件测试工具。 我们做单元测试也主要关注四个方面：组件渲染、状态变化、事件响应、网络请求。

10.2.1 Jest

Jest 是 Facebook 发布的一个开源的、基于 Jasmine 框架的 JavaScript 单元测试工具，支持断言、仿真、快照测试、测试覆盖率报告等。Jest 作为一款测试框架，拥有测试框架该有的一套体系、丰富的断言库，并且大多数 API 与过去熟知的测试框架 Jasmine、Mocha 等基本保持一致，譬如常用的 expect、test(it)、toBe 等 API 都是非常方便、常用的方法。Jest 内部使用 Jasmine 作为基础，在其上进行封装，尤其是 Snapshot 这个特色功能，非常适合 React 项目的测试。测试的方法论，可以根据自己的喜好实践，如 TDD 和 BDD（Jest 对这两者都支持）。React 官方也推荐使用 Jest 作为测试引擎。Jest 的突出特征如下：

（1）支持 Snapshot 组件快照

Jest 通过借用 react-test-renderer 库的 renderer 获得 React 组件渲染成的 React 树，调用 toJSON 接口格式化，再使用 Jest 的 expect(tree).toMatchSnapshot()将快照与上一次的快照做对比，首次生成的某个测试案例的快照将会被保存下来，以后每次运行时都会与上一次对比，如果发现不匹配就会抛出错误，需要开发者查看差异是否是合法的需要更新的内容。这种方式对比 React 组件渲染后的内容，能够非常高效地找出静态 UI 的差别，维护稳定性，并且 Jest 能够对 React 树进行快照或对别的序列化数值快速编写测试，提供快速更新的用户体验。

（2）支持多线程运行测试案例，速度快

Jest 虚拟化了 JavaScript 的环境，能模拟浏览器，并且并行执行。Jest 支持多线程运行测试案例，这个特性在实际项目过程中能够成倍地缩小测试用例执行时间、较大地提升运行效率。

（3）可配置的 coverage reports

Jest 已内置测试覆盖报告特性。若需要生成测试覆盖报告，则无须再下载其他依赖，通过命令 jest --coverage 就能生成 coverage 文件夹保存，方便分析，生成的覆盖报告如图 10-3 所示。

All files

8.08% Statements 549/6794　6.51% Branches 343/5267　7.48% Functions 118/1578　19.05% Lines 546/2866

File		Statements		Branches		Functions		Lines	
actions		0%	0/1465	0%	0/975	0%	0/438	0%	0/262
app		0%	0/49	0%	0/18	0%	0/8	0%	0/43
components		11.27%	62/550	7.17%	32/446	9.85%	13/132	19.62%	62/316
components/act-branch-config		0%	0/50	0%	0/47	0%	0/11	0%	0/11
components/act-root-config		0%	0/54	0%	0/47	0%	0/11	0%	0/14
components/add-module		0%	0/38	0%	0/33	0%	0/9	0%	0/9
components/add-role		0%	0/33	0%	0/31	0%	0/7	0%	0/8
components/branch-select		0%	0/30	0%	0/31	0%	0/7	0%	0/5
components/card-module		0%	0/28	0%	0/27	0%	0/6	0%	0/4
components/data-edit-modal		34.18%	108/316	32.99%	65/197	39.39%	26/66	34.29%	108/315
components/data-table		61.08%	102/167	68.64%	81/118	31.43%	11/35	65.36%	100/153
components/data-table-item		79.7%	106/133	75.9%	63/83	84.21%	32/38	79.7%	106/133
components/group-select		0%	0/29	0%	0/31	0%	0/7	0%	0/4
components/keyword-filter		0%	0/32	0%	0/33	0%	0/8	0%	0/3
components/module-batch-opt		52.17%	12/23	0%	0/2	20%	2/10	52.17%	12/23

图 10-3　测试覆盖率分析

（4）强大的 mocking 库

通过 Jest 可以得到 mock 值、函数、文件多种类型。

①mock 函数：mock 的函数可以被追踪到。`const handler = jest.fn()`的作用是后续可以通过 `expect(handler).toBeCalled()` 检查 mock 的函数是否如期被调用，`expect(handler).toBeCalledWith('arg')`检查 mock 的函数调用时传入的参数是否是`arg`等。

②mock 文件：例如 css、image 等与逻辑测试无关的资源文件可以在配置时就被 mock 掉，`"\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$": "<rootDir>/spec/__mocks__/fileMock.js","\\.(css|scss)$": "<rootDir>/spec/__mocks__/styleMock.js",`，这样`import`时则不会引入这些与测试逻辑无关的文件。

③time mock：`setTimeout, setInterval, clearTimeout, clearInterval`默认是被 mock 的，这样就不会在执行时真的等待这些函数获取的时间参数，影响测试执行速度，但是内部特殊的 mock 让其依旧能保证异步进行和执行顺序。

（5）内置 jsDom，提供了 DOM 依存的环境

Jest 配合 enzyme，enzyme 可以在 jsDom 里渲染出虚拟 DOM，然后我们可以操作 enzyme，进行交互测试。依旧有 window、document 等对象，但是无法往这些虚拟 DOM 中插入 script 标签进行其他资源文件的加载。

此外，Jest 是模块化、可扩展和可配置的，支持异步代码测试，如 promises 和 async/await 等。

10.2.2　Enzyme

Enzyme 是由 AirBnb 团队发布和维护的测试实用程序库，提供了一个更好的、高级的 API 来处理测试中的 React 组件。

Enzyme 提供了几种方式来将 React 组件渲染成真实的 DOM，提供了类似 jQuery 的 API 来获取 DOM。Enzyme 提供了 simulate 函数来模拟事件触发，提供接口来获取组件的 state 和 props 并且能对其进行操作。

Enzyme 实质上是 react-test-renderer 的封装，react-test-renderer 的 API 非常不友好，但是 Enzyme 开发的 API 跟 jQuery 一致，如图 10-4 所示。

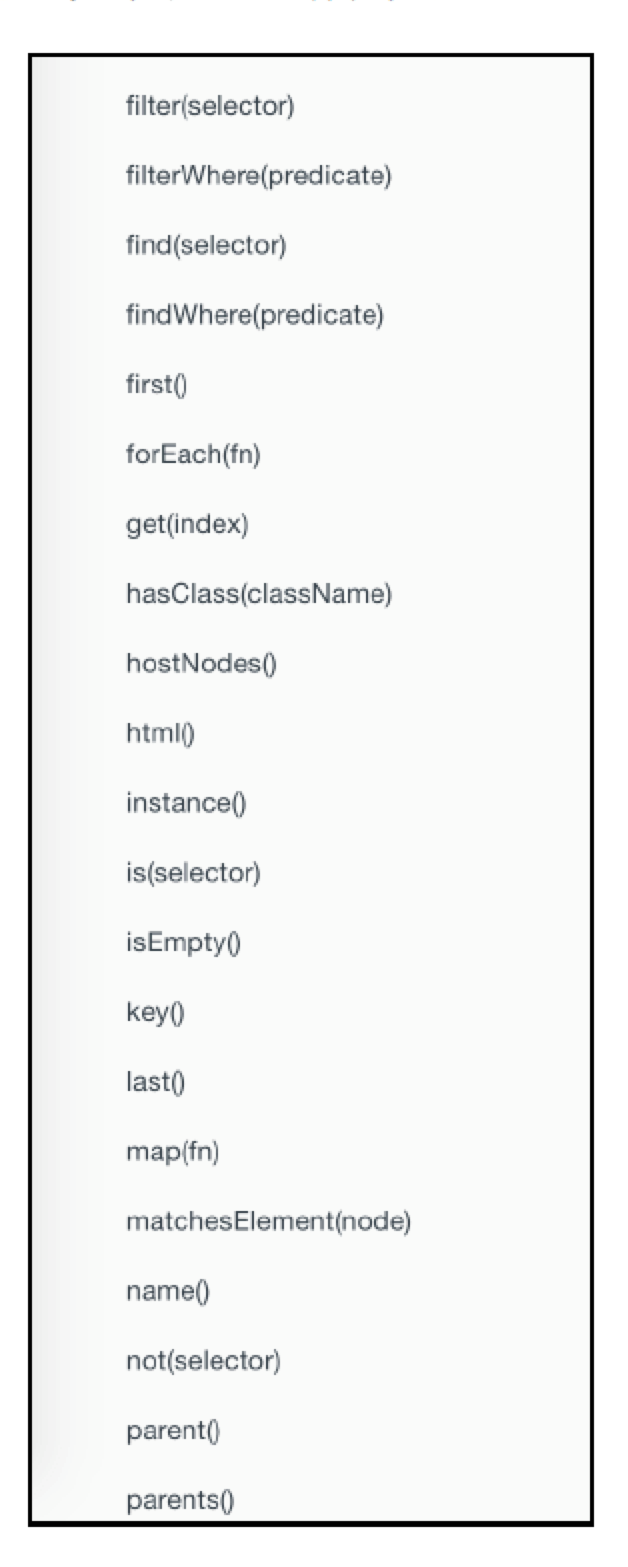


图 10-4　Enzyme 的 API 示例

还可以接入 enzyme-to-json，这样在将 Enzyme 生成的 React 树生成 snapshot 时就可以使用 toJson 进行格式化，提高生成的 snapshot 的可读性，代替了 react-test-renderer 的 toJSON 接口。

10.3 动手测试我们的代码

10.3.1 使用 Jest 测试

【示例 10-1　Jest 测试】

（1）首先，使用 yarn 安装 Jest：

```
yarn add --dev jest
```

或使用 npm 安装 Jest ：

```
npm install --save-dev jest
```

（2）还需安装测试所需要的依赖：

```
npm install -D babel-jest babel-core babel-preset-env regenerator-runtime
```

babel-jest、 babel-core、 regenerator-runtime、babel-preset-env 这几个依赖是为了让我们可以使用 ES6 的语法特性进行单元测试。对于 ES6 提供的 import 导入模块的方式，Jest 本身是不支持的。

（3）在项目的根目录下添加.babelrc 文件，并在文件中添加如下内容：

```
{
  "presets": ["env"]
}
```

（4）将 package.json 文件中 script 的 test 值修改为 jest：

```
"scripts": {
  "test": "jest"
}
```

（5）接下来，创建 src 和 test 目录及相关文件，在项目根目录下创建 src 目录，并在 src 目录下添加 index.js 文件，在项目根目录下创建 test 目录，并在 test 目录下创建 index.test.js 文件。Jest 会自动找到项目中所有使用.spec.js 或.test.js 文件命名的测试文件并执行，通常我们在编写测试文件时遵循的命名规范为“测试文件的文件名 = 被测试模块名 + .test.js”，例如被测试模块为 index.js，对应的测试文件将命名为 index.test.js。

（6）在 src/functions.js 中创建被测试的代码，让我们从写一个两个数相加的示例函数开始。首先，创建一个 sum.js 文件，并添加代码：

```
export default {
  sum(a, b) {
    return a + b;
  }
}
```

（7）在 test/sum.test.js 文件中创建测试用例：

```
import utils  from '../src/sum;

test('sum(2 + 2) 等于 4', () => {
  expect(utils.sum(2, 2)).toBe(4);
})
```

（8）此时，运行 npm run test，Jest 会在终端打印出如下信息：

```
> jest

 PASS  test/sum.test.js
  ✓ sum(2 + 2) 等于 4 (5ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.745s
Ran all test suites.
```

如上所示，我们成功地写了第一个 Jest 测试，使用 expect 和 toBe 来测试两个值完全相同。Jest 为我们提供了 expect 函数来包装被测试的方法并返回一个对象，该对象中包含一系列的匹配器来让我们更方便地进行断言，上面的 toBe 函数即为一个匹配器。下面介绍几种常用的 Jest 断言，其中会涉及多个匹配器。

（1）.not

.not 修饰符允许我们测试结果不等于某个值的情况，这和英语的语法几乎完全一样，很好理解。

【示例 10-2　.not 测试】

```
//utils.test.js
import utils from '../src/utils

test('sum(2, 2) 不等于 5', () => {
  expect(utils.sum(2, 2)).not.toBe(5);
})
```

（2）.toEqual()

.toEqual 匹配器会递归地检查对象所有属性和属性值是否相等，所以如果要进行应用类型的比较时，需要使用.toEqual 匹配器而不是.toBe。

【示例 10-3　.toEqual()测试】

```
// utils.js
export default {
  getAuthor() {
    return {
      name: 'LITANGHUI',
      age: 24,
    }
  }
}
```

```
// functions.test.js
import uitls  from '../src/utils;

test('getAuthor()返回的对象深度相等', () => {
  expect(utils.getAuthor()).toEqual(utils.getAuthor());
})

test('getAuthor()返回的对象内存地址不同', () => {
  expect(utils.getAuthor()).not.toBe(utils.getAuthor());
})
```

（3）.toHaveLength

.toHaveLength 可以很方便地用来测试字符串和数组类型的长度是否满足预期。

【示例 10-4　. toHaveLength 测试】

```
// utils.js
export default {
  getIntArray(num) {
    if (!Number.isInteger(num)) {
      throw Error('"getIntArray"只接受整数类型的参数');
    }

    let result = [];
    for (let I = 0, len = num; I < len; i++) {
      result.push(i);
    }

    return result;
  }
}

// utils.test.js
import utils  from '../src/utils;

test('getIntArray(3)返回的数组长度应该为 3', () => {
  expect(utils.getIntArray(3)).toHaveLength(3);
})
```

（4）.toThrow

.toThrow 能够让我们测试被测试方法是否按照预期抛出异常，但是在使用时需要注意的是：我们必须使用一个函数将被测试的函数做一个包装，正如下面 getIntArrayWrapFn 所做的

那样，否则会因为函数抛出导致该断言失败。

【示例 10-5 .toThrow 测试】

```
// utils.test.js
import utils  from '../src/utils;

test('getIntArray(3.3)应该抛出错误', () => {
  function getIntArrayWrapFn() {
    functions.getIntArray(3.3);
  }
  expect(getIntArrayWrapFn).toThrow('"getIntArray"只接受整数类型的参数');
})
```

（5）.toMatch

.toMatch 传入一个正则表达式，它允许我们用来进行字符串类型的正则匹配。

【示例 10-6 .toMatch 测试】

```
// utils.test.js
import utils  from '../src/utils;

test('getAuthor().name 应该包含"li"这个姓氏', () => {
  expect(utils.getAuthor().name).toMatch(/li/i);
})
```

下面通过一个完整示例来学习如何测试异步函数。

【示例 10-7 测试异步函数】

这里我们使用最常用的 HTTP 请求库 axios 来进行请求处理。首先安装 axios：

```
npm install axios
```

编写 HTTP 请求函数，代码如下：

```
// utils.js
import axios from 'axios';

export default {
  fetchUser() {
    return axios.get('http://jsonplaceholder.typicode.com/users/1')
      .then(res => res.data)
      .catch(error => console.log(error));
  }
}

// utils.test.js
```

```
import utils  from '../src/utils;

test('fetchUser() 可以请求到一个含有 name 属性值为 Leanne Graham 的对象', () => {
  expect.assertions(1);
  return utils.fetchUser()
    .then(data => {
      expect(data.name).toBe('Leanne Graham');
    });
})
```

在上述代码中，我们调用了 expect.assertions(1)，能确保在异步的测试用例中有一个断言会在回调函数中被执行。这在进行异步代码的测试中十分有效。

使用 async 和 await 精简异步代码：

```
test('fetchUser() 可以请求到一个用户名字为 Leanne Graham', async () => {
  expect.assertions(1);
  const data =  await functions.fetchUser();
  expect(data.name).toBe('Leanne Graham')
})
```

10.3.2　使用 Emzyme 测试

（1）首先安装所需依赖：

```
npm install jest --save-dev
npm install enzyme --save-dev
npm install enzyme-to-json --save-dev
```

（2）接着在 package.json 中添加基础配置，例如：

```
"jest": {
  "moduleFileExtensions": [
    "js",
    "jsx",
    "json"
  ],
  "testRegex": ".*\\.spec\\.js$",
  "collectCoverageFrom": [
    "src/**/*.{js,jsx}",
    "src/**/**/*.{js,jsx}",
    "!**/node_modules/**"
  ],
  "moduleNameMapper": {

"\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac
```

```
|oga)$": "<rootDir>/spec/__mocks__/fileMock.js",
      "\\.(css|scss)$": "<rootDir>/spec/__mocks__/styleMock.js"
    },
    "setupFiles": [
      "<rootDir>/spec/setup.js"
    ]
  }
```

- moduleFileExtensions 描述被测试的文件里依赖的文件后缀。配置文件后缀后，依赖的时候只需写需要的文件名即可，工具会根据配置去找相应后缀的文件。
- testRegex 描述正则表示的测试文件。测试文件的目录有很多种规划：①单独放在测试文件夹。②放在原文件夹，和原文件相同的位置。若组件都放在测试文件夹，而业务代码的测试还紧挨着原文件放置，则可通过文件名进行匹配，可将所有测试文件格式都设为 xxx.spec.js。
- collectCoverageFrom 描述生成测试覆盖报告时检测的覆盖文件，源文件代码都存放于 src 下的 js 和 jsx 文件中，但是要去除 node_modules 下的所有文件，因为其会被作为依赖在原文件里处处引入。
- moduleNameMapper 描述 mock 了图片、CSS、字体、音视频文件。
- setupFiles 为配置文件。在运行测试案例代码之前，Jest 会先运行这里的配置文件来初始化所指定的测试环境。setupFiles 是非常有用的配置，可以解决很多问题。在这里可以为全局的 window 或 global 对象绑定内容，配置代码所需的运行环境。例如，在 setup.js 中可配置全局的 react、mock localStorage 等。随着测试的进行，后续还会在这里添加其他配置。

```
import React from 'react';
if (typeof window !== 'undefined') {
  window.React = React;
  window.localStorage = ( function storageMock() {
    var storage = {};
    return {
      setItem: function(key, value) {
        storage[key] = value || '';
      },
      getItem: function(key) {
        return key in storage ? storage[key] : null;
      },
      removeItem: function(key) {
        delete storage[key];
      },
      get length() {
        return Object.keys(storage).length;
      },
      key: function(i) {
```

```
        var keys = Object.keys(storage);
        return keys[i] || null;
      }
    }
  })()
}
```

（3）至此，测试环境就搭好了，接着开始编写单元测试代码。所需要测试的组件或者函数必须从原文件里导出来，测试文件才能引用到，对于原文件的依赖，只要不是被 mock 的文件，都会真实地加载执行。下面给出一个 React 组件测试的结构示例：

```
import React from 'react';
import { render, mount, shallow } from 'enzyme';
import toJson from 'enzyme-to-json';
import { component } from 'component_path';


describe('component test', () => {

  it('test one aspect', () => {

  });
})
```

基于 Jest 和 Emzyme 的特性，针对 React 项目，可以进行以下几类测试。

1. 使用 snapshot 测试组件 UI

snapshot 可以测试组件的渲染结果是否符合预期（预期就是指上一次录入保存的结果），toMatchSnapshot 方法会对比这次将要生成的结果与上次的区别。snapshot 的测试案例形如调用这个组件，传入依赖的 props。例如，对 antd 的 ToolTip 组件进行测试：

```
import { Tooltip } from 'antd';
import { render } from 'enzyme';
import toJson from 'enzyme-to-json';

describe('FileUploadInput render', () => {
  // 基础使用测试
  it('basic use', () => {
    const wrapper = render(
      <Tooltip title="prompt text">
        <span>Tooltip will show when mouse enter.</span>
      </Tooltip>
    );
    expect(toJson(wrapper)).toMatchSnapshot();
```

```
    })
    // 测试箭头指向中心
    it('use arrowPointAtCenter', () => {
      const wrapper = render(
        <div>
          <Tooltip placement="topLeft" title="Prompt Text">
            <Button>Align edge / 边缘对齐</Button>
          </Tooltip>
          <Tooltip placement="topLeft" title="Prompt Text" arrowPointAtCenter>
            <Button>Arrow points to center / 箭头指向中心</Button>
          </Tooltip>
        </div>
      );
      expect(toJson(wrapper)).toMatchSnapshot();
    })

    it('use placement',() => {
      // 快照
      const wrapper = render(<div>
            <div style={{ marginLeft: 60 }}>
              <Tooltip placement="topLeft" title={text}>
                <a href="#">TL</a>
              </Tooltip>
              <Tooltip placement="top" title={text}>
                <a href="#">Top</a>
              </Tooltip>
              <Tooltip placement="topRight" title={text}>
                <a href="#">TR</a>
              </Tooltip>
            </div>
            <div style={{ width: 60, float: 'left' }}>
              <Tooltip placement="leftTop" title={text}>
                <a href="#">LT</a>
              </Tooltip>
              <Tooltip placement="left" title={text}>
                <a href="#">Left</a>
              </Tooltip>
              <Tooltip placement="leftBottom" title={text}>
                <a href="#">LB</a>
              </Tooltip>
            </div>
            <div style={{ width: 60, marginLeft: 270 }}>
              <Tooltip placement="rightTop" title={text}>
```

```
            <a href="#">RT</a>
          </Tooltip>
          <Tooltip placement="right" title={text}>
            <a href="#">Right</a>
          </Tooltip>
          <Tooltip placement="rightBottom" title={text}>
            <a href="#">RB</a>
          </Tooltip>
        </div>
        <div style={{ marginLeft: 60, clear: 'both' }}>
          <Tooltip placement="bottomLeft" title={text}>
            <a href="#">BL</a>
          </Tooltip>
          <Tooltip placement="bottom" title={text}>
            <a href="#">Bottom</a>
          </Tooltip>
          <Tooltip placement="bottomRight" title={text}>
            <a href="#">BR</a>
          </Tooltip>
        </div>
      </div>)
    expect(toJson(wrapper)).toMatchSnapshot();
  })
})
```

需要注意的是，一个足够健壮的测试应该覆盖所有的渲染情况。例如，如果向组件传入不同的 props 参数值，组件会渲染出不同的结果，这时必须编写多个测试用例，以便覆盖所有的渲染结果。回顾 ToolTip 组件，其中 placement 参数表示箭头方向，arrowPointAtCenter 参数表示将 tooltip 的箭头位于指示框中间位置。所以测试用例中应该加上这些内容：

```
it('use arrowPointAtCenter', () => {
  const wrapper = render(
    <div>
      <Tooltip placement="topLeft" title="Prompt Text">
        <Button>Align edge / 边缘对齐</Button>
      </Tooltip>
      <Tooltip placement="topLeft" title="Prompt Text" arrowPointAtCenter>
        <Button>Arrow points to center / 箭头指向中心</Button>
      </Tooltip>
    </div>
  );
  expect(toJson(wrapper)).toMatchSnapshot();
})
```

```
  it('use placement',() => {
    const wrapper = render(<div>
         <div style={{ marginLeft: 60 }}>
           <Tooltip placement="topLeft" title={text}>
             <a href="#">TL</a>
           </Tooltip>
           <Tooltip placement="top" title={text}>
             <a href="#">Top</a>
           </Tooltip>
           <Tooltip placement="topRight" title={text}>
             <a href="#">TR</a>
           </Tooltip>
         </div>
         <div style={{ width: 60, float: 'left' }}>
           <Tooltip placement="leftTop" title={text}>
             <a href="#">LT</a>
           </Tooltip>
           <Tooltip placement="left" title={text}>
             <a href="#">Left</a>
           </Tooltip>
           <Tooltip placement="leftBottom" title={text}>
             <a href="#">LB</a>
           </Tooltip>
         </div>
         <div style={{ width: 60, marginLeft: 270 }}>
           <Tooltip placement="rightTop" title={text}>
             <a href="#">RT</a>
           </Tooltip>
           <Tooltip placement="right" title={text}>
             <a href="#">Right</a>
           </Tooltip>
           <Tooltip placement="rightBottom" title={text}>
             <a href="#">RB</a>
           </Tooltip>
         </div>
         <div style={{ marginLeft: 60, clear: 'both' }}>
           <Tooltip placement="bottomLeft" title={text}>
             <a href="#">BL</a>
           </Tooltip>
           <Tooltip placement="bottom" title={text}>
             <a href="#">Bottom</a>
           </Tooltip>
           <Tooltip placement="bottomRight" title={text}>
```

```
            <a href="#">BR</a>
          </Tooltip>
        </div>
      </div>)
  expect(toJson(wrapper)).toMatchSnapshot();
})
```

2. 使用 Jest 和 Enzyme 测试 DOM 交互

Enzyme 有 3 种渲染方式：render、mount、shallow。

render 采用的是第三方库 Cheerio 的渲染，渲染结果是普通的 HTML 结构。对于 snapshot，使用 render 比较合适。

shallow 和 mount 对组件的渲染结果不是 HTML 的 DOM 树，而是 React 树，如果 Chrome 装了 React Devtool 插件，渲染结果就是在 React Devtool 下查看的组件结构，而 render 函数的结果是在 element 中查看的结果。这些只是渲染结果上的差别，更大的差别是 shallow 和 mount 的结果是被封装的 ReactWrapper，可以进行多种操作。例如，利用 find()、parents()、children()等选择器进行元素查找，利用 state()、props()进行数据查找，利用 setState()、setprops()操作数据，利用 simulate()模拟事件触发。

shallow 只渲染当前组件，只能对当前组件做断言；mount 会渲染当前组件以及所有子组件，对所有子组件也可以做上述操作。一般交互测试都会关心到子组件，都需要使用 mount，但是 mount 耗时更长，内存占用更多。如果没必要操作和断言子组件，可以使用 shallow。

利用 simulate()接口模拟事件进行交互测试，实际上是通过触发事件绑定函数来模拟事件的触发。触发事件后，判断 props 上特定函数是否被调用、传参是否正确、组件状态是否发生预料之中的修改、某个 DOM 节点是否存在以及是否符合期望。

```
import React from 'react';
import { render, mount } from 'enzyme';
import { renderToJson } from 'enzyme-to-json';
import { Table } from 'antd';

describe('Table.pagination', () => {
  const columns = [{
    title: 'Name',
    dataIndex: 'name',
  }];

  const data = [
    { key: 0, name: 'Jack' },
    { key: 1, name: 'Lucy' },
    { key: 2, name: 'Tom' },
    { key: 3, name: 'Jerry' },
  ];
```

```
const pagination = { pageSize: 2 };
// 创建表格
function createTable(props) {
  return (
    <Table
      columns={columns}
      dataSource={data}
      pagination={pagination}
      {...props}
    />
  );
}
// 表格查找
function renderedNames(wrapper) {
  return wrapper.find('TableRow').map(row => row.props().record.name);
}

it('paginate data', () => {
  const wrapper = mount(createTable());
  // 表格内容测试
  expect(renderedNames(wrapper)).toEqual(['Jack', 'Lucy']);
  wrapper.find('Pager').last().simulate('click');
  expect(renderedNames(wrapper)).toEqual(['Tom', 'Jerry']);
});
```

通过触发最后一页的 click 事件达到页面改变，去判断 table 中渲染的数据是否符合预期。

```
it('repaginates when pageSize change', () => {
  const wrapper = mount(createTable());

  wrapper.setProps({ pagination: { pageSize: 1 } });
  expect(renderedNames(wrapper)).toEqual(['Jack']);
});
```

这是直接用 setProps 操作了 pagination 参数，再去判断 table 中渲染的数据是否符合预期。

```
it('fires change event', () => {
  const handleChange = jest.fn();
  const noop = () => {};
  const wrapper = mount(createTable({
    pagination: { ...pagination, onChange: noop, onShowSizeChange: noop },
    onChange: handleChange,
  }));
```

```
    wrapper.find('Pager').last().simulate('click');

    expect(handleChange).toBeCalledWith(
      {
        current: 2,
        onChange: noop,
        onShowSizeChange: noop,
        pageSize: 2,
      },
      {},
      {}
    );
  });
```

触发 click 事件，断言 handleChange 是否以预期参数被调用。

```
    it('should display pagination as prop pagination changed', () => {
      const wrapper = mount(createTable());
      expect(wrapper.find('.ant-pagination')).toHaveLength(1);
      expect(wrapper.find('.ant-pagination-item')).toHaveLength(2);
      wrapper.setProps({ pagination: false });
      expect(wrapper.find('.ant-pagination')).toHaveLength(0);
      wrapper.setProps({ pagination });
      expect(wrapper.find('.ant-pagination')).toHaveLength(1);
      expect(wrapper.find('.ant-pagination-item')).toHaveLength(2);
      wrapper.find('.ant-pagination-item-2').simulate('click');
      expect(renderedNames(wrapper)).toEqual(['Tom', 'Jerry']);
      wrapper.setProps({ pagination: false });
      expect(wrapper.find('.ant-pagination')).toHaveLength(0);
      wrapper.setProps({ pagination: true });
      expect(wrapper.find('.ant-pagination')).toHaveLength(1);
      expect(wrapper.find('.ant-pagination-item')).toHaveLength(1); // pageSize
will be 10
      expect(renderedNames(wrapper)).toEqual(['Jack', 'Lucy', 'Tom', 'Jerry']);
    });
  });
```

先判断渲染结果中的子节点.ant-pagination、.ant-pagination-item 数量是否符合预期，以达到渲染分页情况是否正确的判断，然后通过 setProps 操作 pagination 参数判断子节点是否符合预期。

从以上案例也可看出，断言时既可以通过获取 props()、state()中的数据来判断是否符合预期，也可以先通过 dom selector 查询找到特定节点再通过 text()接口拿到数据来判断是否符合预期。

3. 功能函数测试

功能函数的测试除了测试普通的工具函数，还需要测试函数调用、传入特定参数的调用，判断函数返回值是否符合预期。需要注意的也是全面性，保证函数的每个逻辑判断都能在所有测试案例跑完后被覆盖到。

这时纯函数和函数式编程的优势就体现出来了——纯函数方便测试。对于使用 Redux 的项目，比较特殊的是 action、reducer 函数的测试。

（1）action

action 其实也是调用函数判断返回值是否符合预期，例如：

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

测试代码为：

```
import * as actions from '../../actions/TodoActions'
import * as types from '../../constants/ActionTypes'

describe('actions', () => {
  it('should create an action to add a todo', () => {
    const text = 'Finish docs'
    const expectedAction = {
      type: types.ADD_TODO,
      text
    }
    expect(actions.addTodo(text)).toEqual(expectedAction)
  })
})
```

异步 action 的测试需要借助第三方库 configureMockStore 进行，将 redux-thunk 异步中间件传入 configureMockStore，获得封装后的 store.dispatch 来派发 action。测试代码为：

```
  import configureMockStore from 'redux-mock-store'
  import thunk from 'redux-thunk'
  import * as actions from '../../actions/TodoActions'
  import * as types from '../../constants/ActionTypes'
  import nock from 'nock'

  const middlewares = [ thunk ]
  const mockStore = configureMockStore(middlewares)
```

```
describe('async actions', () => {
  afterEach(() => {
    nock.cleanAll()
  })

  it('creates FETCH_TODOS_SUCCESS when fetching todos has been done',()=>{
    nock('http://example.com/')
      .get('/todos')
      .reply(200, { body: { todos: ['do something'] }})

    const expectedActions = [
      { type: types.FETCH_TODOS_REQUEST },
      { type: types.FETCH_TODOS_SUCCESS, body: { todos: ['do something']  } }
    ]
    const store = mockStore({ todos: [] })

    return store.dispatch(actions.fetchTodos())
      .then(() => { // 异步 actions 的返回
        expect(store.getActions()).toEqual(expectedActions)
      })
  })
})
```



异步的测试案例一定要返回异步 promise，return 不能丢，这样才能让 Jest 明白这是一个异步过程，它才会去等待异步执行结果，否则会立即触发 expect 断言导致测试失败。

（2）mock

很多对于测试没有价值的东西可以 mock 掉，不会影响测试的准确性。对于一个 React 组件，测试这个组件时并不需要关心注入它的 action 函数的行为，所要做的是特定场景下 action 函数被正确调用，调用结果则是 action 函数的单元测试该验证的事。例如，可以定义 const handler = jest.fn()，后续可以通过 expect(handler).toBeCalled()检查 mock 的函数是否如期被调用，expect(handler).toBeCalledWith('arg')检查 mock 函数调用时传入的参数是否是 arg 等。

诸如 css、image 等与逻辑测试无关的资源文件可以在 package.json 中的配置里被统一 mock 掉：

```
"jest": {
  "moduleNameMapper": {

"\\.(jpg|jpeg|png|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|m4a|aac|oga)$": "<rootDir>/spec/__mocks__/fileMock.js",
    "\\.(css|scss)$": "<rootDir>/spec/__mocks__/styleMock.js",
```

```
    "^component_path$": "<rootDir>/src/components",
    "^root_path$": "<rootDir>/src"
  }
}
```

import 时就不会真的引入这些对于测试无用的文件了。

我们并不想在测试时真的去发一个请求，一方面是会耗时很长，另一方面可能会用测试数据污染线上数据库，所以我们需要 mock 掉真正的 HTTP 请求，模拟返回值。不用担心不准确，只用保证请求时的参数符合期望、mock 的返回值按预期编写就好，至于这些请求是否真的能返回这些结果交给接口测试处理即可。手动 mock 请求效率实在太低，所以我们需要一个组件直接拦截掉所有请求，无须改变项目代码，传入所需 mock 的返回值，组件封装成请求的返回值后，直接返回给测试代码。

首先封装一个统一的 mock 方法：

```
const fetchMock = require('fetch-mock');
import { HOST } from '../src/util/api.js';

export function mockRequest(path,res){
  let reg = new RegExp(`${HOST}${path}.*`);
  return fetchMock.get( reg, {
    body: {
      status:{
        code: "0",
        detail: "成功",
        msg: "success"
      },
      result: res
    },
    status: 200
  })
}
```

调用方法如下：

```
import { mockRequest } from '../mockRequest.js';
import { multiData } from './mockModuleData.js';

describe('DataEdit render', () => {
  afterEach(fetchMock.restore)
  it('renders DataEdit correctly', () => {
    mockRequest('/pageModule/getData', multiData);
    return wrapper.node.fetchData(moduleInfo.moduleId).then(()=>{
      expect(toJson(wrapper)).toMatchSnapshot();
    })
```

```
  });
})
```

10.4 测试之外

10.4.1　PropTypes

PropTypes 定义为组件类自身的属性，用以定义 prop 的类型。在开发模式下，当提供一个不合法的值作为 prop 时，控制台会出现警告；在产品模式下，为了性能考虑应忽略 PropTypes。例如：

```
import React from 'react'
  class Son extends React.Component {
  render(){
    return (<div style ={{padding:30}}>
      {this.props.number}
      <br/>
      {this.props.array}
      <br/>
      {this.props.boolean.toString()}
    </div>)
   }
}
class Father extends React.Component {
   render(){
     return (<Son
       number = {'1'}
       array = {'[1,2,3]'}
       boolean = {'true'}
     />)
  }
}
```

在这个示例中，我们通过从父组件向子组件传递属性，原本试图通过数值、阵列和布尔这三个属性分别向子组件中传递一个数字、数组和一个布尔型数值，但是由于失误把它们都写成了字符串，虽然渲染是正常的，但是这可能会导致接下来调用一些方法的时候发生错误，而系统并不提供任何提示。

下面让我们给它加上 PropTypes 的类型检测：

```
import React from 'react'
import PropTypes from 'prop-types';
```

```
class Son extends React.Component{
   render(){
      return (<div style ={{padding:30}}>
        {this.props.number}
        <br/>
        {this.props.array}
        <br/>
        {this.props.boolean.toString()}
      </div>)
    }
}
Son.propTypes = {
        number:PropTypes.number,
        array:PropTypes.array,
        boolean:PropTypes.bool
}
class Father extends React.Component{
    render(){
        return (<Son
        number = {'1'}
        array = {'[1,2,3]'}
        boolean = {'true'}
      />)
    }
}
```

然后我们就能看到报的错误了，而且这个时候报的错误包括错误的道具属性名称、错误的变量类型、属性所在的组件名称、预期正确的变量类型、错误代码的位置以及其他更详细的信息，如图 10-5 所示。



图 10-5　通过 ProTypes 进行类型检测

PropTypes 能用来检测全部数据类型的变量，包括基本类型的字符串、布尔值、数字以及引用类型的对象、数组、函数，甚至还有 ES6 新增的符号类型。React.PropTypes 输出一系列的验证器，用以确保收到的数据是合法的。如下示例记录了不同的验证器：

```
MyComponent.propTypes = {
```

```
// 可以声明 prop 是特定的 JS 基本类型
// 默认情况下这些 prop 都是可选的
optionalArray:PropTypes.array,
optionalBool: PropTypes.bool,
optionalFunc: PropTypes.func,
optionalNumber: PropTypes.number,
optionalObject: PropTypes.object,
optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,

// 任何可以被渲染的事物：numbers, strings, elements or an array
// (or fragment) containing these types.
optionalNode: PropTypes.node,

// A React element.
optionalElement: PropTypes.element,

// 声明一个 prop 是某个类的实例，用到了 JS 的 instanceOf 运算符
optionalMessage: PropTypes.instanceOf(Message),

// 用 enum 来限制 prop 只接受特定的值
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// 指定的多个对象类型中的一个
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// 指定类型组成的数组
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// 指定类型的属性构成的对象
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// 一个指定形式的对象
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// 你可以用以上任何验证器链接‘isRequired’，以确保 prop 不为空
```

```
    requiredFunc: PropTypes.func.isRequired,

    // 不可空的任意类型
    requiredAny: PropTypes.any.isRequired,

    // 自定义验证器，如果验证失败，必须返回一个 Error 对象
    // 不要直接使用 console.warn 或者 throw，这些在 oneOfType 中都没用
    customProp: function(props, propName, componentName) {
      if (!/matchme/.test(props[propName])) {
        return new Error(
          'Invalid prop `' + propName + '` supplied to' +
          ' `' + componentName + '`. Validation failed.'
        );
      }
    },

    // 你也可以为 arrayOf 和 objectOf 提供一个验证器
    // 如果验证失败，它也应该返回一个 Error 对象
    // 在 array 或者 object 中，验证器对于每个 key 都会被调用 The first two
    // 验证器的前两个 arguments 是 array 或者 object 自身以及当前的 key 值
    customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
      if (!/matchme/.test(propValue[key])) {
        return new Error(
          'Invalid prop `' + propFullName + '` supplied to' +
          ' `' + componentName + '`. Validation failed.'
        );
      }
    })
  };
```

可以使用 React.PropTypes.element 指定仅可以将单一子元素作为子节点传递给组件。

```
  class MyComponent extends React.Component {
    render() {
      // This must be exactly one element or it will warn.
      const children = this.props.children;
      return (
        <div>
          {children}
        </div>
      );
    }
  }
```

```
MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};
```

通过赋值特殊的 defaultProps 属性，可以为 props 定义默认值：

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

// Renders "Hello, Stranger":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

如果父组件没有为 this.props.name 传值，那么 defaultProps 会给其一个默认值。PropTypes 的类型检测是在 defaultProps 解析之后发生的，因此也会对默认属性 defaultProps 进行类型检测。

10.4.2　Flow

Flow 本质上只是一个检查工具，并不会自动修正代码中的错误，也不会强制说你没有按照它的警告消息修正，只是不会让程序运行而已。当然，并没有要求什么时候一定要用这类工具，只是这种做法可以让你的代码更具强健性并提高阅读性，也可以直接避去很多不必要的数据类型使用上的问题。这种开发方式目前在许多框架与函数库项目或是以 JavaScript 应用为主的开发团队中都已经是必用的了。

JavaScript 是一种弱（动态）数据类型的语言，弱（动态）数据类型代表在代码中，变量或常量会自动依照赋值变更数据类型，而且类型种类也很少。这是直译式脚本语言的常见特性，但既有可能是优点也有可能是很大的缺点。优点是容易学习与使用，缺点是开发者经常会因为赋值或传值的类型错误而造成不如预期的结果。有些时候在使用框架或函数库时，如果没有仔细看文件，或者文件写得不清楚，也容易造成误用的情况。

（1）首先，通过 npm 安装 Flow：

```
npm install --save-dev flow-bin
```

（2）初始化项目：

```
flow init
```

通常在代码的最顶部一行加入声明（没加的话，Flow 工具是不会进行检查的），以下两种格式都可以：

```
// @flow
```

或

```
/* @flow */
```

可以使用下面的命令指令来进行代码检查：

```
flow check
```

例如，如下代码：

```
function foo(x) {
  return x + 10
}

foo('Hello!')
```

利用 Flow 类型的定义方式，可以改写为下面这样的代码：

```
// @flow

function foo(x: number): number {
  return x + 10
}

foo('hi')
```

可以看到，在函数的传参以及函数的圆括号(())后面的两个地方都加了:number 标记，代表这个传参会限定为数字类型，并且返回值也只允许是数字类型。

当使用非数字类型的值作为传入值时，就会出现由 Flow 工具发出的警告消息：

```
message: '[flow] string (This type is incompatible with number See also: function call)'
```

如果是要允许多种类型，也是很容易加标记的。例如，某一函数可以使用布尔或数字类型的入参，出参的类型可以是数字或字符串：

```
// @flow
```

```
function foo(x: number | boolean): number | string {
  if (typeof x === 'number') {
    return x + 10
  }
  return 'x is boolean'
}

foo(1)
foo(true)
foo(null)  // 这一行有类型错误消息
```

如果在多人协同开发某个有规模的 JavaScript 应用时，这种类型的输出输入问题就会很常见。如果利用 Flow 工具进行检查，就可以避免掉许多不必要的类型问题。

Flow 支持的原始数据类型包括以下几项：

- boolean
- number
- string
- null
- void

其中的 void 类型就是 JS 中的 undefined 类型。需要注意的是，在 JS 中，undefined 与 null 的值会相等但类型不同，意思是做值相等比较时，比如 undefined == null 时会为 true。有时在运行期间的检查时，可能会用值相等比较而不是严格的相等比较来检查这两个类型的值。

10.4.3　TypeScript

TypeScript 是 JavaScript 的超集并且能够编译输出为纯粹的 JavaScript，安装方法如下：

```
npm install -g typescript
```

我们需要一个 tsconfig.json 文件来告诉 ts-loader 如何编译 TypeScript 代码。在当前根目录下创建 tsconfig.json 文件，并添加如下内容：

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  },
```

```
    "include": [
      "./src/**/*"
    ]
  }
```

其中：

- outDir：输出目录。
- sourceMap：把 ts 文件编译成 js 文件的时候，同时生成对应的 sourceMap 文件。
- noImplicitAny：如果为 true，当 TypeScript 编译器无法推断出类型时，它就会生成 JavaScript 文件，但是会报告一个错误。为了找到错误，还是设置为 true 比较好。
- module：代码规范，也可以选 amd。
- target：转换成 es5。
- jsx：TypeScript 具有三种 JSX 模式，即 preserve、react 和 react-native。这些模式只在代码生成阶段起作用（类型检查并不受影响）。在 preserve 模式下，生成代码中会保留 JSX，以供后续的转换操作使用（比如：Babel）。另外，输出文件会带有.jsx 扩展名。react 模式会生成 React.createElement，在使用前不再需要进行转换操作，输出文件的扩展名为.js。 react-native 相当于 preserve，也保留了所有的 JSX，但是输出文件的扩展名是.js。我们这里因为不会用 Babel 再转，所以用 react 就行。
- include：需要编译的目录。

在编辑器中，将下面的代码输入到 greeter.ts 文件里：

```
function greeter(person) {
    return "Hello, " + person;
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

上面虽然使用了.ts 扩展名，但是这段代码仅仅是 JavaScript 而已。 在命令行上，运行 TypeScript 编译器：

```
tsc greeter.ts
```

输出结果为一个 greeter.js 文件，包含了和输入文件中相同的 JavaScript 代码。一切准备就绪，我们可以运行这个使用 TypeScript 写的 JavaScript 应用了。

接下来看看 TypeScript 工具带来的高级功能。

【示例 10-8　: string 类型注解】

给 person 函数的参数添加: string 类型注解：

```
function greeter(person: string) {
    return "Hello, " + person;
```

```
}

let user = "Jane User";

document.body.innerHTML = greeter(user);
```

TypeScript 里的类型注解是一种轻量级的为函数或变量添加约束的方式。在这个例子里，我们希望 greeter 函数接收一个字符串参数。然后尝试把 greeter 的调用改成传入一个数组：

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

重新编译，会看到产生了一个错误：

```
error TS2345: Argument of type 'number[]' is not assignable to parameter of type
'string'.
```

类似地，尝试删除 greeter 调用的所有参数。TypeScript 会告诉你使用非期望个数的参数调用了这个函数。在这两种情况中，TypeScript 提供了静态的代码分析，可以用来分析代码结构和提供的类型注解。

要注意的是尽管有错误，greeter.js 文件还是被创建了。就算你的代码里有错误，也仍然可以使用 TypeScript，但在这种情况下 TypeScript 会警告你代码可能不会按预期执行。

接着，创建目录：

```
mkdir src && cd src
mkdir components && cd components
```

在此文件夹下添加一个 Hello.tsx 文件，代码如下：

```
import * as React from 'react';

export interface Props {
  name: string;
  enthusiasmLevel?: number;
}

export default class Hello extends React.Component<Props, object> {
  render() {
    const { name, enthusiasmLevel = 1 } = this.props;
    if (enthusiasmLevel <= 0) {
      throw new Error('You could be a little more enthusiastic. :D');
```

```
    }
    return (
      <div className="hello">
        <div className="greeting">
          Hello {name + getExclamationMarks(enthusiasmLevel)}
        </div>
      </div>
    );
  }
}

function getExclamationMarks(numChars: number) {
  return Array(numChars + 1).join('!');
}
```

接下来，在 src 下创建 index.tsx 文件，代码如下：

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import Hello from "./components/Hello";

ReactDOM.render(
  <Hello name="TypeScript" enthusiasmLevel={10} />,
  document.getElementById('root') as HTMLElement
);
```

我们还需要一个页面来显示 Hello 组件。在根目录创建一个名为 index.html 的文件，代码如下：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>demo</title>
</head>
<body>
  <div id="root"></div>
  <script src="./dist/bundle.js"></script>
</body>
</html>
```

在根目录下创建一个名为 webpack.common.config.js 文件，并添加如下内容：

```
module.exports = {
  entry: "./src/index.tsx",
  output: {
```

```
    filename: "bundle.js",
    path: __dirname + "/dist"
  },

  devtool: "source-map",
  resolve: {
    extensions: [".ts", ".tsx", ".js", ".json"]
  },
  module: {
    rules: [
      { test: /\.tsx?$/, loader: "ts-loader" },
      { enforce: "pre", test: /\.js$/, loader: "source-map-loader" }
    ]
  },

  plugins: [
  ],
};
```

在根目录下运行以下命令：

```
webpack --config webpack.common.config.js
```

打开 index.html 就能进行调试了。

第 11 章 性能优化

前端资源包括 HTML、CSS、Javascript、Image、Video 等多种不同类型的资源。前端性能优化的目的是让页面加载得更快、对用户的操作响应得更及时、给用户提供更为友好的体验，同时优化能够减少页面请求数或者减小请求所占带宽，能够节省可观的资源。总之，提升用户体验是核心目标。前端针对不同类型的资源分别都有不同的性能优化方式。

11.1 不要过早优化

性能优化思路大致如图 11-1 所示。



图 11-1　性能优化流程

不要将性能优化的精力浪费在对整体性能提高不大的代码上，而对性能有关键影响的部分，优化并不嫌早，因为对性能影响最关键的部分往往涉及解决方案核心、决定整体的架构，将来要改变的时候牵扯更大。

React 利用虚拟 DOM 来提升渲染性能。虽然每一次页面更新都是对组件的重新渲染，但是并不是将之前的渲染内容全部抛弃重来，借助虚拟 DOM，React 能够计算出对 DOM 树的最少修改，这就是 React 默认情况下渲染都很迅速的秘诀。

虽然虚拟 DOM 能够将每次 DOM 操作量减少到最小，但计算和比较虚拟 DOM 依然是一个复杂的过程。当然，如果能够在开始计算虚拟 DOM 之前就判断渲染的结果不会有变化，那么就可以不进行虚拟 DOM 计算和比较，速度就会更快。

shouldComponentUpdate 优化在图中去掉了许多凹坑，并减少了整体渲染时间。用同样的方法可以避免更多的重绘（例如避免重绘 sidebar、操作按钮、没有变化的表头和页码），但是别到处都加 shouldComponentUpdate ，在简单组件上执行 shouldComponentUpdate 方法有时比仅渲染组件要耗时。也别在应用的早期使用，这将过早地进行优化。随着应用的壮大，我们会发现组件上的性能瓶颈，此时添加 shouldComponentUpdate 逻辑能保持快速地运行。

11.2 React 性能查看工具

在讲性能优化之前，我们需要先来了解一下如何查看 React 加载组件时所耗费时间的工具，在 React 16 版本之前，我们可以使用 React Perf 来查看。Perf 是 React 官方提供的性能分析工具。Perf 最核心的方法莫过于 Perf.printWasted(measurements)，该方法会列出那些没必要的组件渲染。在很大程度上，React 的性能优化就是除掉这些无谓的渲染。

可以在 Chrome 中先安装 React Perf 扩展，然后在入口文件或者 Redux 的 store.js 中加入相应的代码：

```
import {createStore, combineReducers, applyMiddleware, compose} from 'redux';

import {reducer as todoRedcer} from './todos';
import {reducer as filterReducer} from './filter';

import Perf from 'react-addons-perf'; // 性能工具

const win = window;
win.Perf = Perf;
// reducer 组合
const reducer = combineReducers({
  todos: todoReducer,
  filter: filterReducer
});
// 中间件组合
const middlewares = [];
if (process.env.NODE_ENV !== 'production') {
  middlwwares.push(require('redux-immutable-state-invariant')());
}
// 创建 store
const storeEnhancers = compose(
  applyMiddleware(...middlewares),
  (win && win.devToolsExtension) ? win.devToolsExtension() : (f) => f,
)
```

在 React 16 版本中，直接在 url 后加上“?react_pref”，就可以在 Chrome 浏览器的 performance 中利用 User Timeing 来查看组件的加载时间，如图 11-2 所示。

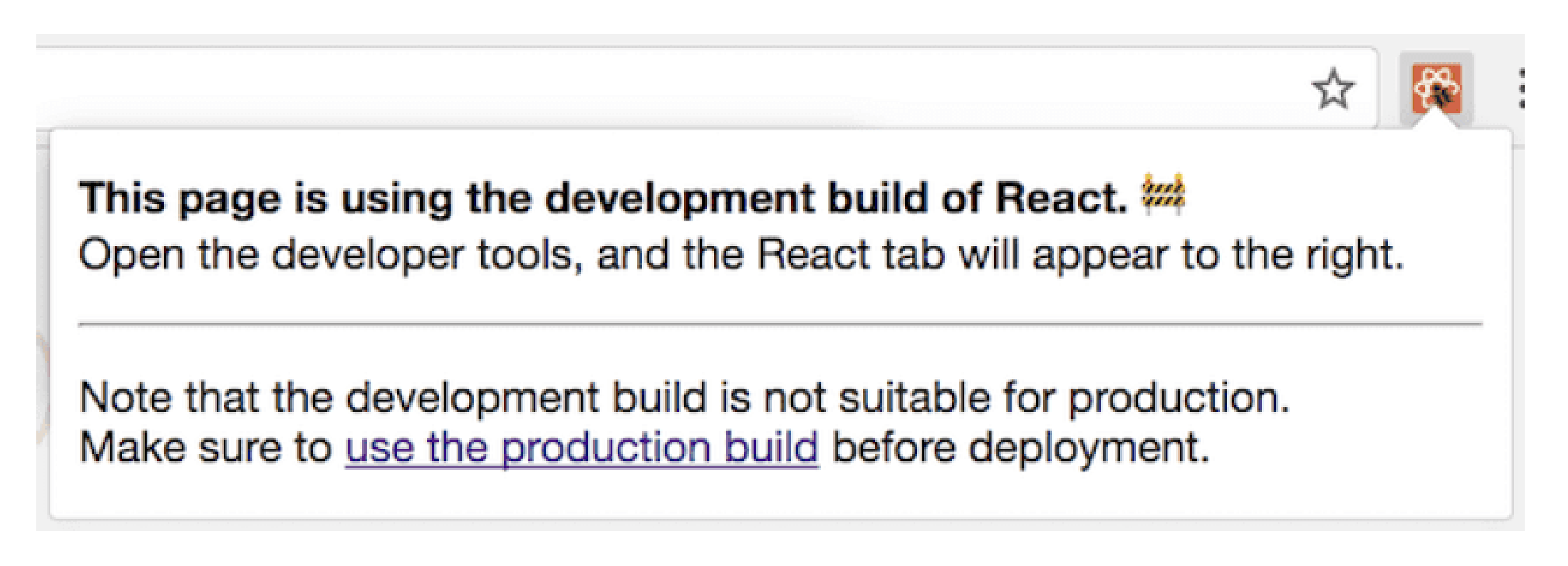


图 11-2 Chrome React 插件

11.3 React 优化手段

11.3.1 单个 React 组件性能优化

一般来说，要尽可能少地在 render 函数中做操作。

render 里面尽量减少新建变量和 bind 函数，尽量减少所需传递参数的数量，尽可能地保持 props 和 state 简单和精简。

例如，单击事件有 3 种实现方法：

- 第一种是在构造函数中绑定 this。
- 第二种是在 render()函数里面绑定 this。
- 第三种就是使用箭头函数。

这 3 种方法都能实现，构造函数每一次渲染的时候只会执行一遍。在 render()函数里面绑定 this，在每次 render()的时候都会重新执行一遍函数。采用第三种方法，每一次 render()的时候都会生成一个新的箭头函数，即使两个箭头函数的内容是一样的。React 使用浅层比较判断是否需要触发 render，简单来说就是通过===来判断，如果 state 或者 prop 的类型是字符串或者数字，只要值相同，那么浅层比较就会认为其相同；如果前者的类型是复杂的对象（对象是引用类型），那么浅层比较只会判断这两个 prop 是不是同一个引用，如果不是，哪怕这两个对象中的内容完全一样，也会被认为是两个不同的 prop。

我们给组件 Foo 中名为 style 的 prop 赋值：

```
<Foo style={{ color:"red" }}
```

使用这种方法，因为每一次渲染都会产生一个新对象传递给 style，所以每一次渲染都会被认为是 style 这个 prop 发生了变化。

那么我们应该如何改进呢？如果想要让 React 渲染的时候认为前后对象类型 prop 相同，就必须保证 prop 指向同一个 JavaScript 对象，例如：

```
const fooStyle = { color: "red" };
```

```
            //确保这个初始化只执行一次，不要放在 render 中，可以放在构造函数中

  <Foo style={fooStyle} />
```

11.3.2　shoudComponentUpdate

React 有一个生命周期函数 shouldComponentUpdate()，这个方法可以根据当前和下一次的 props 和 state 来通知 React 组件是否应该被重新渲染。shouldComponentUpdate 是决定 React 组件什么时候能够不重新渲染的函数，但是这个函数默认的实现方式是简单地返回一个 true。也就是说，默认每次更新的时候都要调用所用的生命周期函数，包括 render 函数，重新渲染。shouldComponentUpdate 生命周期如图 11-3 所示。

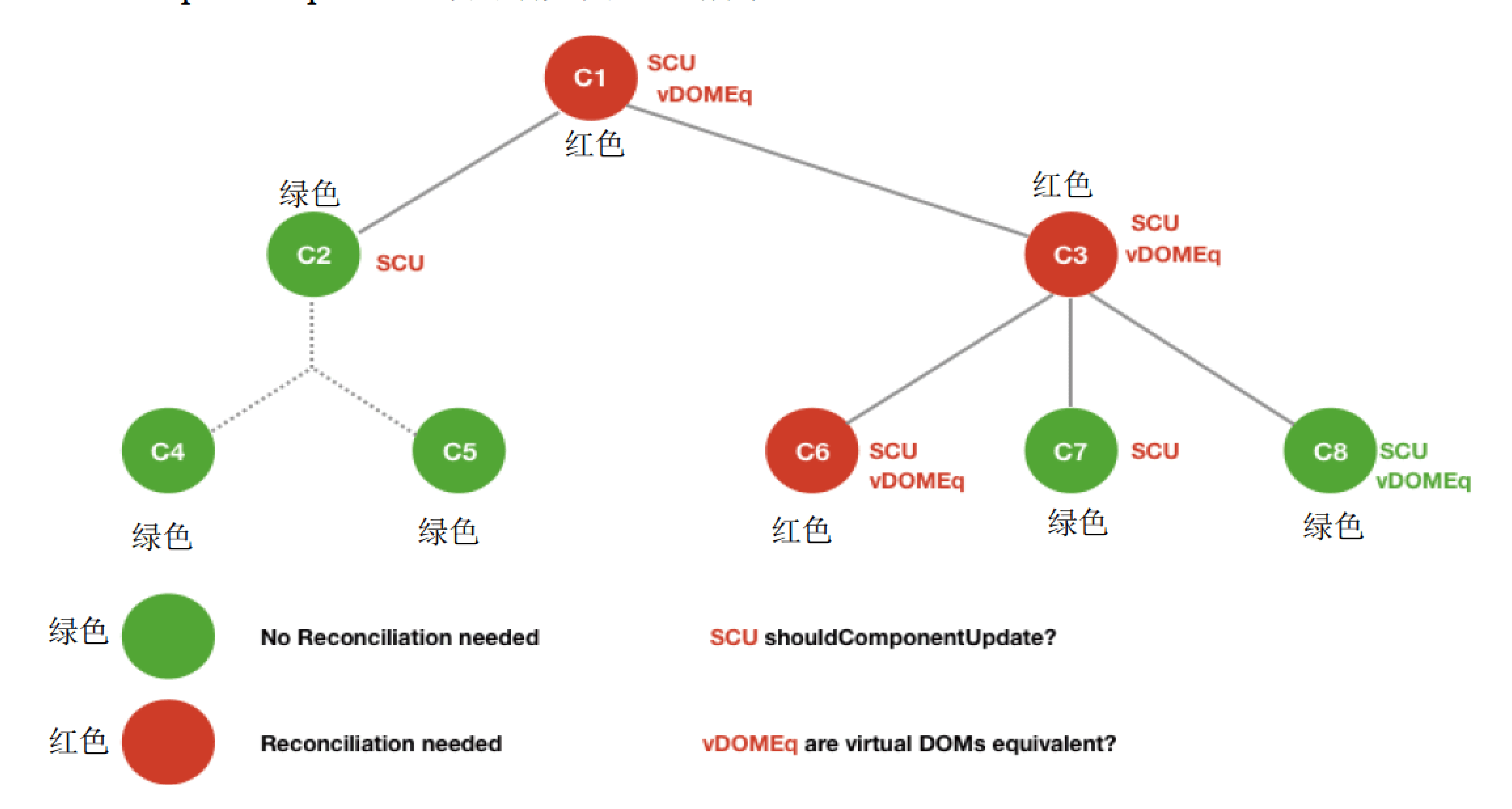


图 11-3　shouldComponentUpdate

其中，SCU 表示 shouldComponentUpdate，绿色表示返回 true（需要更新），红色表示返回 false（不需要更新）；vDOMEq 表示虚拟 DOM 比对，绿色表示一致（不需要更新），红色表示发生改变（需要更新）。根据渲染流程，首先会判断 shouldComponentUpdate(SCU)是否需要更新。如果需要更新，就调用组件的 render 生成新的虚拟 DOM，然后与旧的虚拟 DOM 对比（vDOMEq），如果对比一致就不更新，如果对比不同，就根据最小粒度改变去更新 DOM；如果 SCU 不需要更新，就直接保持不变，同时其子元素也保持不变。

- C1 根节点，绿色 SCU（true），表示需要更新，然后 vDOMEq 红色，表示虚拟 DOM 不一致，需要更新。
- C2 节点，红色 SCU（false），表示不需要更新，所以 C4、C5 均不再进行检查。
- C3 节点同 C1，需要更新。
- C6 节点，绿色 SCU（true），表示需要更新，然后 vDOMEq 红色，表示虚拟 DOM 不一致，更新 DOM。

- C7 节点同 C2。
- C8 节点，绿色 SCU（true），表示需要更新，然后 vDOMEq 绿色，表示虚拟 DOM 一致，不更新 DOM。

开发者必须考虑到需要触发重新渲染的每一种情况，这会导致逻辑复杂化，但是很多情况下会有更好的选择。

React 从 V15 开始包含了一个 PureComponent 类，它可以被用来构建组件。React.PureComponent 声明了它自己的 shouldComponentUpdate()方法，自动对当前和下一次的 props 和 state 做一次浅对比。在大多数情况下，React.PureComponent 是比 React.Component 更好的选择。在创建新组件时，首先尝试将其构建为纯组件，只有组件的功能需要时才使用 React.Component。

还有一些细节的优化点：

- 慎用 {...this.props}，只传递 component 需要的 props，传得太多或者层次传得太深，都会加重 shouldComponentUpdate 里面的数据比较负担，因此慎用 spread attributes（<Component {...props} />））。
- ::this.handleChange()，将该方法的 bind 于 constructor 中进行定义，即 this.handleChange.bind(this,id)。
- 复杂的页面不要全部写在一个组件里面，尽量拆分。
- 尽量使用 const element。
- map 中需要添加 key，并且 key 不要使用 index。
- 尽量少用 setTimeout 或不可控的 refs、DOM 操作。
- props 和 state 的数据尽可能简单明了，扁平化。
- 使用 return null 而不是 CSS 的 display:none 来控制节点的显示隐藏，保证同一时间页面的 DOM 节点尽可能少。

11.3.3 immutable（ImmutableJS）

我们也可以在 shouldComponentUpdate()中使用 deepCopy 和 deepCompare 来避免不必要的 render()，但 deepCopy 和 deepCompare 一般都是非常耗性能的。

Immutable Data 就是一旦创建就不能再被更改的数据。对 immutable 对象的任何修改或添加、删除操作都会返回一个新的 immutable 对象。

immutable 实现的原理是 Persistent Data Structure（持久化数据结构），也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变。同时为了避免 deepCopy 把所有节点都复制一遍带来的性能损耗，immutable 使用了 StructuralSharing（结构共享），即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其他节点则进行共享。

immutable 提供了简洁高效的判断数据是否变化的方法，只需用===和 is 比较就能知道是否需要执行 render()，而这个操作几乎零成本，所以可以极大地提高性能。修改后的 shouldComponentUpdate 如下：

```
import { is } from 'immutable';

shouldComponentUpdate: (nextProps = {}, nextState = {}) => {
  const thisProps = this.props || {}, thisState = this.state || {};
  // 比较属性个数
  if (Object.keys(thisProps).length !== Object.keys(nextProps).length ||
      Object.keys(thisState).length !== Object.keys(nextState).length) {
    return true;
  }
  // 对比每个属性的值，不相等的时候返回 true，表示需要更新
  for (const key in nextProps) {
    if (!is(thisProps[key], nextProps[key])) {
      return true;
    }
  }
  // 比较每个属性，且属性存在
  for (const key in nextState) {
    if (thisState[key] !== nextState[key] || !is(thisState[key],
nextState[key])) {
      return true;
    }
  }
  return false;
}
```

immutable 是一个完全独立的库，无论基于什么框架都可以用，弥补了 JavaScript 没有不可变数据结构的问题。由于是不可变的，因此可以放心地对对象进行任意操作。在 React 开发中，频繁操作 state 对象或是 store ，配合 ImmutableJS 会更快、更安全、更方便。

immutable 的优点如下：

（1）immutable 降低了 Mutable 带来的复杂度

可变（Mutable）数据耦合了 Time 和 Value 的概念，使数据很难被回溯。

（2）节省内存

immutable.js 使用了 Structure Sharing，会尽量复用内存，甚至以前使用的对象也可以再次被复用，没有被引用的对象会被垃圾回收。

```
import { Map} from 'immutable';
let a = Map({
  select: 'users',
  filter: Map({ name: 'Cam' })
})
let b = a.set('select', 'people');
```

```
a === b; // false
a.get('filter') === b.get('filter'); // true
```

Undo/Redo、Copy/Paste，甚至可以实现时间旅行这些功能。

因为每次数据都是不一样的，只要把这些数据放到一个数组里储存起来，想回退到哪里就拿出对应数据，很容易开发出撤销、重做这种功能。

（3）并发安全

传统的并发非常难做，因为要处理各种数据不一致的问题，因此发明了各种锁来解决。使用了 immutable 之后，数据天生是不可变的，就不需要并发锁了。

（4）拥抱函数式编程

immutable 本身就是函数式编程中的概念，纯函数式编程比面向对象更适用于前端开发。因为只要输入一致，输出必然一致，这样开发的组件更易于调试和组装。

ImmutableJS 中有几个重要的 API，具有如下：

（1）fromJS()

fromJS()是最常用的将原生 JS 数据转换为 ImmutableJS 数据的转换方法。使用方式类似于 JSON.parse()，接收两个参数：json 数据和 reviver 函数。

在不传递 reviver 函数的情况下，默认将原生 JS 的 Array 转为 List、Object 转为 Map：

```
// 常见
const t1 = Immutable.fromJS({a: {b: [10, 20, 30]}, c: 40});
console.log(t1);

// 不常用
const t2 = Immutable.fromJS({a: {b: [10, 20, 30]}, c: 40},function(key, value){
    // 定制转换方式下这种就是将 Array 转换为 List、Object 转换为 Map
    const isIndexed = Immutable.Iterable.isIndexed(value);
    return isIndexed ? value.toList() : value.toOrderedMap();
    // true, "b", {b: [10, 20, 30]}
    // false, "a", {a: {b: [10, 20, 30]}, c: 40}
    // false, "", {"": {a: {b: [10, 20, 30]}, c: 40}}
});
console.log(t2);
```

（2）toJS()

先来看官网的一段话：immutable 数据应该被当作值而不是对象，值是表示该事件在特定时刻的状态。这个原则对理解不可变数据的适当使用是最重要的。为了将 immutable.js 数据视为值，就必须使用 immutable.is()函数或.equals()方法来确定值相等，而不是确定对象引用标识的===操作符。

toJS()就是用来对两个 immutable 对象进行值比较的。使用方式类似于 Object.is(obj1,obj2)，接收两个参数：

```
const map1 = Immutable.Map({a:1, b:1, c:1});
const map2 = Immutable.Map({a:1, b:1, c:1});

// 两个不同的对象
console.log(map1 === map2); // false
// 进行值比较
console.log(Immutable.is(map1, map2)); // true

// 不仅仅只能比较 ImmutableJS 类型的数据
console.log(Immutable.is(undefined, undefined)); // true
console.log(Immutable.is(null, undefined)); // false
console.log(Immutable.is(null, null)); // true
console.log(Immutable.is(NaN, NaN)); // true

// 区别于 Object.is
console.log(Object.is(0, -0) ,Immutable.is(-0, 0)); // false , true
```

（3）Map

Map 数据类型对应原生 Object 数组，是最常用的数据结构之一，循环时无序（orderedMap 有序），对象的 key 可以是任意值，具体看下面的例子。

```
console.log(Map().set(List.of(1), 'list-of-one').get(List.of(1)));
console.log(Map().set(NaN, 'NaN').get(NaN));
console.log(Map().set(undefined, 'undefined').get(undefined));
console.log(Map().set(null, 'null').get(null));
```

OrderedMap 是 Map 的变体，除了具有 Map 的特性外，还具有顺序性。当开发者遍历 OrderedMap 的实例时，遍历顺序为该实例中元素的声明、添加顺序。OrderedMap 比非有序 Map 更昂贵，并且可能消耗更多的内存。如果真要求遍历有序，建议使用 List。

（4）List

List 数据类型对应原生 Array 数组，和原生数组最大的区别是不存在空元素，如不存在[, , , ,]：

```
console.log(List([,,,,]).toJS());
 // [undefined, undefined, undefined, undefined]
```

11.4 性能优化小结

本章从代码级两个粒度对 React 性能优化的方式做了一些介绍，这些方法基本上都是前端开发人员在开发的过程中可以借鉴和实践的，除此之外，完整的前端优化还应该包括很多其他的途径，例如 CDN、Gzip、多域名、无 Cookie 服务器等。

第 12 章 Hooks

在 React Conf 2018 中提出了 Hooks 这个概念，并在大会上宣布 React v16.7.0-alpha(内测)将引入 Hooks，所以我们有必要了解 Hooks，以及由此引发的疑问。本章让我们一起来详细讨论 Hooks 解决的问题以及使用方法。

12.1 为什么引入 Hooks

React Hooks 用来解决在项目中长期使用和维护 React 过程中遇到的一些难以避免的问题，其中一个核心问题是如何实现业务逻辑代码分离，从而实现组件内部相关业务逻辑的复用。

一般情况下，我们都是通过组件和自上而下传递的数据流将我们页面上的大型 UI 拆分为独立的小型 UI 组件，实现组件的重用。但是，在实际的项目中，经常会发现，组件的逻辑是有状态的，无法将逻辑提取到函数组件中，从而导致复杂组件难以实现重用。这类问题在处理动画和表单时尤为常见。当我们在组件中连接外部的数据源，并希望在组件中执行更多其他操作时，我们就会把组件做得过于复杂。类似的问题有：

（1）难以跨组件复用包含状态的逻辑。

难以重用和共享组件中与状态相关的逻辑，从而产生很多代码里较大的组件。React 没有提供一种将可复用的行为复用到组件上的方式（如 redux 的 connect 方法）。render props 和 高阶组件的出现就是为了解决逻辑复用的问题。但是这些模式都要求开发人员重新构造已有的组件，这种重构可能会非常麻烦。在很多典型的 React 组件中，可以在 React DevTool 里看到这些组件被层层叠叠的 providers、 consumers、高阶组件、render props 和其他抽象层包裹。因此，React 需要一些更好的底层元素来复用包含状态的组件逻辑。

（2）逻辑复杂的组件难以开发与维护。

我们在刚开始构建项目中所需的组件时，组件初期往往相对简单。然而随着开发的进展，组件会变得越来越大、逻辑越来越多、代码越来越混乱，各种逻辑在组件中散落的到处都是。当我们的组件需要处理多个互不相关的 local state 时，每个生命周期钩子中都包含了一堆互不相关的逻辑。例如常常会在 componentDidMount 或 componentDidUpdate 中拉取数据，同时 compnentDidMount 方法可能又包含一些不相干的逻辑，如设置事件监听，之后需要在 componentWillUnmount 中清除。最终的结果是强相关的代码被分离，反而是不相关的代码被

组合在了一起。

在许多情况下，我们也不太可能将这些组件分解成更小的组件，因为逻辑到处都是，测试起来也非常困难。这也是许多开发者喜欢将 React 与单独的状态管理库结合使用的原因之一。然而，这通常会引入太多的抽象，需要在不同的文件之间跳转，并且使得重用组件更加困难。

（3）类组件中的 this 增加学习成本。

类组件在基于现有工具的优化上存在些许问题，React 中功能和类组件的区别以及何时使用每种组件都会导致有经验的 React 开发人员之间的分歧。开发者必须了解 this 如何在 JavaScript 中工作，这与它在大多数语言中的工作方式非常不同，开发者必须记住绑定事件处理程序。

（4）由于业务变动，函数组件不得不改为类组件。

刚开始编写 React 代码时较为常用函数式组件，函数式组件代码简洁、编写效率快。然而，随着业务的变化，后面会发现很多以前写的组件需要修改，会发现很多地方都需要生命周期和状态来进行渲染优化，需要将大量的函数式组件修改为 class 组件。

在这种情况下，Hooks 是这些问题的一种解决方案。Hooks 允许我们将组件内部的逻辑组织成为一个可复用的隔离模块。ReactHooks 需要实现清晰明确的数据流和组成形式，既可以复用组件内的逻辑，也不会出现 HOC 带来的层层嵌套，更不会出现 Mixin 的弊端。例如，使用类组件实现单击按钮次数+1 组件通常会有如下代码实现：

```
import React from 'react';

class Example extends React.Component {
   constructor(props) {
      super(props);
      this.state = {count: 0};
      // 单击事件声明
      this.clickBtn = this.clickBtn.bind(this);
}
// 单击事件
   clickBtn = () => {
      this.setState({
         count: this.state.count + 1;
      });
}
// 渲染
   return (
      <div>
         <p>You clicked {this.state.count} times</p>
         <button onClick={this.clickBtn}>
            Click me
         </button>
```

```
        </div>
    );
}
```

使用 React Hooks 之后，代码可以修改为：

```
// 单击计数器
import { useState } from 'react';

function Example() {
  // 引入 count 和 setCount
  const [count, setCount] = useState(0);

  // 在 useEffect 中处理异步事件
  useEffect(() =({
    document.title = `You clicked ${count} times`;
  });
  // 渲染
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

其中 Hooks 提供的 API 可以大幅减少 React 函数组件的代码量，在下一节中详细介绍 Hooks 的使用方法。

12.2 Hooks 的使用方法

Hooks 让我们的函数组件拥有了类似类组件的特性。

12.2.1 useState

直接来看一个 useState 的例子。

【示例 12-1 useState】

```
import { useState } from 'react';
```

```
function Example() {
  // 引入 count 和 setCount
  const [count, setCount] = useState(0);
  // 引入 age 和 setAge
  const [age, setAge] = useState(42);
  // 引入 fruit 和 setFruit
  const [fruit, setFruit] = useState('Mac');
  // 引入 todos 和 setTodos
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useState 的参数就是 state 的初始值。useState 返回的值中第一个参数是以前的 state，第二个参数是 setState。过去只有一个 state，现在可以自由命名 state，更加直观，如上代码示例中的 age 和 setAge、fruit 和 setFruit。useState 方法可以为我们的函数组件带来 local state，它接收一个用于初始 state 的值，返回一对变量：

```
const [count, setCount] = useState(0);

// 等价于
var const = useState(0)[0];          // 该 state
var setConst = useState(0)[1];       // 修改该 state 的方法
```

12.2.2　useEffect

每当 React 更新之后，就会触发 useEffect（在第一次 render 和每次 update 后触发）。useEffect 可以利用我们组件中的 local state 进行一些带有副作用的操作，例如：

```
useEffect(() = ({
  document.title = `You clicked ${count} times`;
});
```

useEffect 中还可以通过传入第二个参数来决定是否执行里面的操作来避免一些不必要的性能损失，只要第二个参数数组中的成员值没有改变，就会跳过此次执行。如果传入一个空数组[ ]，那么该 effect 只会在组件 mount 和 unmount 时期执行，上述代码可以修改为：

```
useEffect(() = ({
  document.title = `You clicked ${count} times`;
}, [count]);                    // 如果 count 没有改变，就跳过此次执行
```

useEffect 中还可以通过让函数返回一个函数来进行一些清理性的操作，例如取消订阅：

```
useEffect(() = ({
  api.subscribe(theId);
  return () = ({
      api.unsubscribe(theId)     //clean up
  }
});
```

useEffect 会在组件 mount 和 unmount 以及每次重新渲染的时候都执行，也就是会在 componentDidMount、componentDidUpdate、componentWillUnmount 这三个生命周期中执行。清理函数会在前一次 effect 执行后、下一次 effect 将要执行前以及 Unmount 时期执行。

12.2.3 useReducer

利用 React Hooks 可以轻松创建一个 Redux 机制。

【示例 12-2 useReducer】

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  const [add, dispatch] = useReducer(todosReducer);
  // ...
}

function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);
  // 触发
  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

这个自定义 Hook 的 value 部分当作 redux 的 state，setValue 部分当作 redux 的 dispatch，合起来就是一个 redux。而 react-redux 的 connect 与 Hook 调用方式一样：

```
// 一个 Action
function useTodos() {
  const [todos, dispatch] = useReducer(todosReducer, []);
```

```
    // 处理单击事件
    function handleAddClick(text) {
      dispatch({ type: "add", text });
    }

    return [todos, { handleAddClick }];
  }

  // 绑定 Todos 的 UI
  function TodosUI() {
    const [todos, actions] = useTodos();
    return (
      <div>
        {todos.map((todo, index) => (
          <div>{todo.text}</div>
        ))}
        <button onClick={actions.handleAddClick}>Add Todo</button>
      </div>
    );
  }
```

需要注意的是，React Hooks 只提供状态处理方法，不会持久化状态。除此之外，还提供了许多有用的 Hooks：

- useCallback
- useMemo
- useContext
- useRef
- useImperativeMethods
- useMutationEffect
- useLayoutEffect

12.2.4　Hooks 使用限制

我们只能在函数组件（functional component）中使用 Hooks，同时也可以在一个组件中使用多组 Hooks，例如：

【示例 12-3　使用多组 Hooks】

```
  function FriendStatusWithCounter(props) {
    const [count, setCount] = useState(0);
    useEffect(() = ({
      document.title = `You clicked ${count} times`;
    });
```

```
    // 定义 isOnline 和 setIsOnline
    const [isOnline, setIsOnline] = useState(null);
    // 在 useEffect 中处理异步方法
    useEffect(() = ({
      API.subscribe(props.friend.id);
      return () = ({
        API.unsubscribe(props.friend.id);
      });
    });

    return isOnline
  }
```

需要注意的是，只能在顶层代码（Top Level）中调用 Hooks，不能在循环或判断语句等里面调用，这是为了让 Hooks 在每次渲染的时候都会按照相同的顺序调用，因为 useState 需要依赖参照第一次渲染的调用顺序来匹配对应的 state，否则 useState 会无法正确返回对应的 state。useState 的源码分析如下：

```
  let globalHooks;
  function useState(defaultValue) {
    // 试图从 globalHooks 全局变量中获取最后一次运行的状态，在调用组件函数之前设置 Map 对象，
    // Map 应该有上次运行后的数据，否则需要创建自己的数据 hookData
    let hookData = globalHooks.get(useState);
    if (!hookData) hookData = { calls: 0, store: [] };
    // hookData.store 是一个数组，用于存储上次调用的 hookData.calls 值，该值用于跟踪
    // 此函数被我们的组件调用的程度
    if (hookData.store[hookData.calls] === undefined)
      // 在第一次调用时应该返回 defaultValue；通过 hookData.store[hookData.calls]，
      // 可以获取调用的最后一个存储值；如果它不存在则必须使用 defaultValue
  hookData.store[hookData.calls] = defaultValue;

    let value = hookData.store[hookData.calls];
    let calls = hookData.calls;
    // setValue 回调用于单击按钮时更新我们的数据值，例如当单击一个按钮，会调用 calls，
    // 因此它知道 setState 函数调用属于哪个 calls。然后使用 hookData.render 回调，
    // 启动所有组件的重新呈现
    let setValue = function(newValue) {
      hookData.store[calls] = newValue;
      hookData.render();
    };
    // hookData.calls 计数器增加
    hookData.calls += 1;
    // hookData 被存储在 globalHooks 变量中，因此组件函数返回后，可通过 render 函数使用
    globalHooks.set(useState, hookData);
```

```
  return [value, setValue];
}
```

12.3 Hooks 实践

对 React 组件来说，Hooks 是一个高效的 API 插件，本节介绍如何在 React 实践中使用 Hook 进行优化代码结构。

12.3.1　与状态有关的逻辑重用

首先，对于与状态有关的逻辑进行重用和共享问题，过去这类问题的一个解决方案是，Render Props 通过 props 接收一个返回 react element 的函数来动态决定自己要渲染的结果：

```
<DataProvider render={data = (
  <h1>Hello {data.target} </h1>;
)}>
```

另外一种解决方案是使用 HOC 生产组件：

```
function generateComponent(WrappedComponent) {
  return class extends React.Component {
    constructor(props) {
      super(props);
    }

    componentDidMount() {
      // 逻辑代码
    }
    componentWillUnmount() {
      // 逻辑代码
    }
    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}
```

这两种方法都会造成组件数量增多，甚至是修改组件树结构，而且有可能出现组件多层嵌套的情况。通过使用 Hooks，可以通过函数来封装与状态有关的逻辑，将这些逻辑从组件中抽取出来。在这个函数中我们既可以使用其他的 Hooks，也可以单独进行测试，例如：

```
import { useState, useEffect } from 'react';
```

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // useEffect 处理
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

如上代码通过 useEffect 封装对 count 的操作。useCount 其实就是一个函数，我们可以在现有的所有其他组件中进行调用：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

第 12.1 节中提到，组件可能会随着项目和业务的发展变得越来越复杂，组件中要处理越来越多的状态，那么在组件的生命周期函数中就会充斥着各种互不相关的逻辑。这里需要引入官方比较复杂的例子，先看基于以前类组件的情况：

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    // 异步
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
```

通过使用 Hook，上述代码可以修改为：

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() = ({
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() = ({
```

```
        ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
        return () = ({
          ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
        };
      });

      function handleStatusChange(status) {
        setIsOnline(status.isOnline);
      }
      // ...
    }
```

通过 Hook 修改后的代码，状态和相关的处理逻辑都可以按照功能进行划分，不必在各个生命周期中进行管理，大大降低了开发和维护的难度。

12.3.2 DOM 操作副作用的修改

在页面中总有一些看上去和组件关系不大的副作用需要处理，例如修改页面标题、监听页面大小变化（组件销毁记得取消监听）、断网时提示（组件嵌套）。例如，修改页面 title，在组件里调用 useDocumentTitle 函数即可设置页面标题，且切换页面时，页面标题重置为默认标题 “我的信息”：

```
useDocumentTitle("个人中心");
```

直接用 document.title 赋值，在销毁时再次给一个默认标题即可，这个简单的函数可以抽象在项目工具函数里，供其他页面组件调用：

```
function useDocumentTitle(title) {
  useEffect(
    () => {
      document.title = title;
      return () => (document.title = "我的信息");
    },
    [title]
  );
}
```

监听页面大小变化、网络是否断开，在组件调用 useWindowSize 时，可以获取页面宽高，并且在浏览器缩放时自动触发组件更新：

```
const windowSize = useWindowSize();
return <div>页面宽度：{windowSize.innerWidth}</div>;
```

通过 window.innerHeight 等 API 直接获取页面宽高，此时可以用 window.addEventListener('resize')

监听页面大小变化，此时调用 setValue 将会触发调用自身的 UI 组件 rerender，最后注意在销毁时 removeEventListener 将注销监听：

```
function getSize() {
  return {
    innerHeight: window.innerHeight,
    innerWidth: window.innerWidth,
    outerHeight: window.outerHeight,
    outerWidth: window.outerWidth
  };
}

function useWindowSize() {
  let [windowSize, setWindowSize] = useState(getSize());

  function handleResize() {
    setWindowSize(getSize());
  }
  // 在这里处理异步
  useEffect(() => {
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, []);

  return windowSize;
}
```

12.3.3　Hooks 互相引用

React Hooks 可以引用其他 Hooks，例如：

【示例 12-4　Hooks 引用其他 Hooks】

```
import { useState, useEffect } from "react";

// 底层 Hooks，返回布尔值
function useFriendStatusBoolean(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
```

```
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

// 上层 Hooks，根据在线状态返回字符串
function useFriendStatusString(props) {
  const isOnline = useFriendStatusBoolean(props.friend.id);

  if (isOnline === null) {
    return "Loading...";
  }
  return isOnline ? "Online" : "Offline";
}

// 使用了底层 Hooks 的 UI
function FriendListItem(props) {
  const isOnline = useFriendStatusBoolean(props.friend.id);
  return (
    <li style={{ color: isOnline ? "green" : "black" }}>{props.friend.name}</li>
  );
}

// 使用了上层 Hooks 的 UI
function FriendListStatus(props) {
  const statu = useFriendStatusString(props.friend.id);
  return <li>{statu}</li>;
}
```

在这个示例代码中，有两个 Hooks：useFriendStatusBoolean 与 useFriendStatusString。useFriendStatusString 是利用 useFriendStatusBoolean 生成的新 Hook，这两个 Hook 可以给不同的 UI（FriendListItem、FriendListStatus）使用，因为两个 Hooks 数据是联动的，所以两个 UI 的状态也是联动的。实现有状态的组件没有渲染，有渲染的组件没有状态：useFriendStatusBoolean 与 useFriendStatusString 是有状态的组件（使用 useState），没有渲染（返回非 UI 的值），这样就可以作为 Custom Hooks 被任何 UI 组件调用；FriendListItem 与 FriendListStatus 是有渲染的组件（返回了 JSX），没有状态（没有使用 useState），这就是一个纯函数 UI 组件。

12.3.4　处理动画

使用 React Hooks 处理动画，可以获取一些具有弹性变化的值，将这些值赋给进度条之类的组件，这样其进度变化就符合某种动画曲线。例如，在某个时间段内获取 0~1 之间的值，可以通过 useRaf(t)获取 t 毫秒内不断刷新的 0~1 之间的数字，期间组件会不断刷新，但刷新频率由 requestAnimationFrame 控制，不会造成 UI 卡顿。

【示例 12-5　使用 Hooks 处理动画】

```
import React, {
  useState,
  useEffect,
  useLayoutEffect,
  useCallback,
  useRef
} from "react";
import { useRaf } from "react-use";
import ReactDOM from "react-dom";

function App() {
  const value = useRaf(5000);

  return <div>{value}</div>;
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

还有一类是异步请求。通过 useAsync 将一个 Promise 拆解为 loading、error、result 三个对象：

```
const { loading, error, result } = useAsync(fetchUser, [id]);
```

在 Promise 的初期设置 loading，结束后设置 result，如果出错则设置 error：

```
export function useAsync(asyncFunction) {
  const asyncState = useAsyncState(options);

  useEffect(() => {
    const promise = asyncFunction();
    asyncState.setLoading();
    promise.then(
      result => asyncState.setResult(result);,
      error => asyncState.setError(error);
    );
```

```
  }, params);
}
```

参考 react-async-hook 完整的实现：

```
import { useEffect, useRef, useState } from "react";

const InitialAsyncState = {
  loading: true,
  result: undefined,
  error: undefined,
};

const defaultSetLoading = (asyncState) => InitialAsyncState;
const defaultSetResult = (result, asyncState) => ({
  loading: false,
  result: result,
  error: undefined,
});
// 默认异常设置
const defaultSetError = (error, asyncState) => ({
  loading: false,
  result: undefined,
  error: error,
});

// 默认选项
const DefaultOptions = {
  setLoading: defaultSetLoading,
  setResult: defaultSetResult,
  setError: defaultSetError,
};
// 标准选项
const normalizeOptions = options => ({
  ...DefaultOptions,
  ...options,
});

// 异步设置状态
const useAsyncState = options => {
  const [value, setValue] = useState(InitialAsyncState);
  return {
    value,
```

```
    setLoading: () => setValue(options.setLoading(value)),
    setResult: result => setValue(options.setResult(result, value)),
    setError: error => setValue(options.setError(error, value)),
  }
};


const useIsMounted = () => {
  const ref = useRef(false);
  useEffect(() => {
    ref.current = true;
    return () => ref.current = false;
  }, []);
  return () => ref.current;
};


const useCurrentPromise = () => {
  const ref = useRef(null);
  return {
    set: (promise => ref.current = promise),
    is: (promise => ref.current === promise),
  };
};


export const useAsync = (asyncFunction, params = [], options) => {

  options = normalizeOptions(options);
  const AsyncState = useAsyncState(options);
  const isMounted = useIsMounted();
  const CurrentPromise = useCurrentPromise();

  // 仅处理 promise 的 result 和 error 状态
  // 如果是最后一个操作，并且 comp 仍然挂载
  const shouldHandlePromise = p => isMounted() && CurrentPromise.is(p);

  const executeAsyncOperation = () => {
    const promise = asyncFunction(params);
    CurrentPromise.set(promise);
    AsyncState.setLoading();
    promise.then(
      result => {
```

```
        if ( shouldHandlePromise(promise) ) {
          AsyncState.setResult(result);
        }
      },
      error => {
        if ( shouldHandlePromise(promise) ) {
          AsyncState.setError(error);
        }
      }
    );
  };

  useEffect(() => {
    executeAsyncOperation();
  }, params);


  const result = {
    ...AsyncState.value,
    execute: () => executeAsyncOperation(),
  };

  return result;
};
```

例如，获取用户信息 redux 的实现：

```
@Connect(...)
class App extends React.PureComponent {
  public componentDidMount() {
    this.props.fetchUser()
  }

  public render() {
    // this.props.userData.isLoading | error | data
  }
}
```

通过 Hooks 封装的 useAsync 则可以改成：

```
// 获取用户信息
const fetchUser = id =>
```

```
  fetch(`xxx`).then(result => {
    if (result.status !== 200) {
      throw new Error("bad status = " + result.status);
    }
    return result.json();
  });
// 封装
function useFetchUser(id) {
  const asyncFetchUser = useAsync(fetchUser, id);
  return asyncUser;
}

function App() {
  const { isLoading, error, data } = useFetchUser();
}
```

12.3.5　模拟生命周期

通过 useMount 拿到 mount 周期才执行的回调函数实现 componentDidMount：

```
useMount(() => {
  // 类似于 componentDidMount
});
```

通过 useUnmount 拿到 unmount 周期才执行的回调函数实现 componentWillUnmount：

```
useUnmount(() => {
  // 类似于 componentWillUnmount
});
```

通过 useUpdate 拿到 didUpdate 周期才执行的回调函数 componentDidUpdate：

```
useUpdate(() => {
  // 类似于 componentDidUpdate
});
```

componentDidUpdate 除了第一次初始化之外，等价于 useMount 的逻辑每次执行。因此采用 mouting flag（判断初始状态）+不加限制参数确保每次 rerender 都会执行即可：

```

const mounting = useRef(true);
useEffect(() => {
  if (mounting.current) {
    mounting.current = false;
  } else {
    fn();
  }
```

```
});
```

12.4 Hooks 小结

本章介绍了 Hooks 如何解决使用 React 开发和维护项目过程中长期存在的问题。Hooks 还处于早期阶段，但是给我们复用组件的逻辑提供了一种很好的解决问题思路，可以在 react-16.7.0-alpha.0 中体验。把 React Hooks 当作更便捷的 renderProps 来使用，虽然写法看上去是内部维护了一个状态，但其实等价于注入、Connect、HOC 或者 renderProps，使用 renderProps 的门槛会大大降低，我们可以抽象大量 Custom Hooks，而不会增加代码的嵌套层级。

React 官方的目标是尽可能快地让 Hooks 覆盖所有的类组件的诉求，但是现在 Hooks 还处于一个非常早的阶段，相关的调试工具、第三方库等都还没有做好对 Hooks 的支持，而且目前也没有可以取代类组件中 getSnapshotBeforeUpdate 和 componentDidCatch 的方案，不过相信后续很快会在 Hooks 中添加这些特性。总的来说，鼓励使用 Hooks，对于已存在的类组件，不必再大规模地去重写。至于 Hooks 是否可以代替 render-props 和 higher-ordercomponents，官方提议在大多数案例下代替。官方表示，Hooks 及 Hooks 的生态会继续完善。

第 13 章 React实战：React+webpack+ES6实现简易笔记本

本章我们将使用在前几章所阐述的相关知识，使用 React 实现简易的笔记本，包括环境配置、前台界面和后台应用，构建单页面应用。

13.1 配置环境

13.1.1 前台准备

本章我们将从零开始构建实现一个笔记本应用。首先从创建数据库开始，然后创建后台服务，最后实现前台用户可操作的界面。

首先创建文件夹，用于同时存放前台应用和后台应用：

```
mkdir notebook_app && cd notebook_app
```

然后我们从前台应用开始着手。本例使用 create-react-app 创建前台项目，使用 create-react-app 提供的初始的 webpack 和 Babel 配置。若尚未安装 create-react-app，则使用如下命令全局安装：

```
npm i -g create-react-app
```

create-react-app 安装完成之后，使用如下命令创建 react 项目：

```
create-react-app client && cd client
```

项目中需要使用 ajax 来发送 get 或 post 请求，使用如下命令安装 axios：

```
npm i -S axios
```

安装完成之后，编辑 App.js 文件，渲染笔记本提示信息，当后台应用创建完成之后我们再继续完善前台界面：

```
// client/src/App.js
import React, { Component } from "react";
```

```
class App extends Component {
  render() {
    return <div>欢迎来到笔记本应用！ :-)</div>;
  }
}

export default App;
```

使用如下命令行启动程序：

```
npm start
```

在浏览器中打开 http://localhost:3000/ ，可以看到如图 13-1 简单的界面，说明前台应用启动成功。

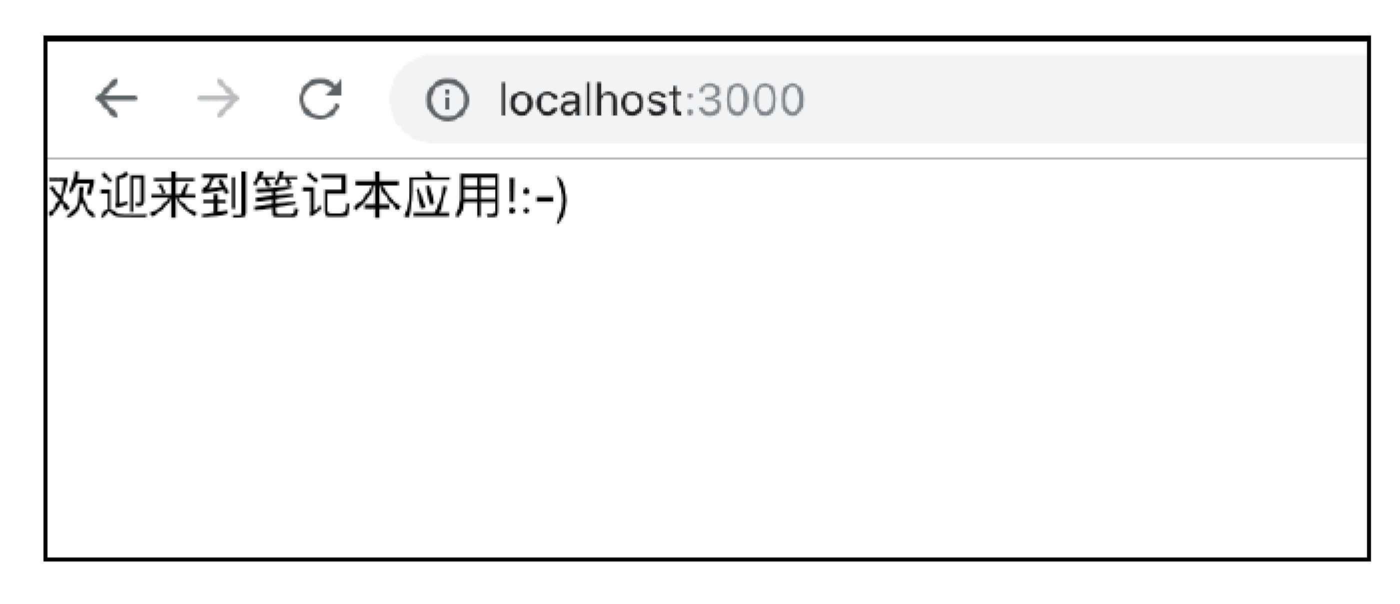


图 13-1　前台界面启动成功

13.1.2　服务端准备

回到笔记本应用的根目录，即 13.1.1 节中创建的 notebook_app 目录下，创建后台应用文件夹，并初始化 package.json：

```
mkdir backend && cd backend
npm init
```

创建后台应用的主程序文件 server.js ，并输入如下代码：

```
// 引入mongoose
const mongoose = require("mongoose");
// 引入express
const express = require("express");
// 引入body-parser
const bodyParser = require("body-parser");
const logger = require("morgan");
const Data = require("./data");

const API_PORT = 3001;
// express 实例
```

```
    const app = express();
    // 路由
    const router = express.Router();

    // 定义 MongoDB 数据库
    const dbRoute = " mongodb://test:test1234@ds37468.mlab.com:
37468/notebook_app";

    // 连接数据库
    mongoose.connect(
      dbRoute,
      { useNewUrlParser: true }
    );

    let db = mongoose.connection;
    /
    db.once("open", () => console.log("connected to the database"));

    // 检测数据库连接是否成功
    db.on("error", console.error.bind(console, "MongoDB connection error:"));
    // 转换为可读的 json 格式
    app.use(bodyParser.urlencoded({ extended: false }));
    app.use(bodyParser.json());
    // 开启日志
    app.use(logger("dev"));

    // 获取数据的方法
    // 用于获取数据库中所有可用数据
    router.get("/getData", (req, res) => {
      Data.find((err, data) => {
        if (err) return res.json({ success: false, error: err });
        return res.json({ success: true, data: data });
      });
    });

    // 数据更新方法
    // 用于对数据库中已有数据进行更新
    router.post("/updateData", (req, res) => {
      const { id, update } = req.body;
      Data.findOneAndUpdate(id, update, err => {
        if (err) return res.json({ success: false, error: err });
        return res.json({ success: true });
      });
```

```
  });

  // 删除数据方法
  // 用于删除数据库中已有数据
  router.delete("/deleteData", (req, res) => {
    const { id } = req.body;
    Data.findOneAndDelete(id, err => {
      if (err) return res.send(err);
      return res.json({ success: true });
    });
  });

  // 添加数据方法
  // 用于在数据库中增加数据
  router.post("/putData", (req, res) => {
    let data = new Data();

    const { id, message } = req.body;

    if ((!id && id !== 0) || !message) {
      return res.json({
        success: false,
        error: "INVALID INPUTS"
      });
    }
    data.message = message;
    data.id = id;
    data.save(err => {
      if (err) return res.json({ success: false, error: err });
      return res.json({ success: true });
    });
  });

  // 对http请求增加 /api 路由
  app.use("/api", router);
  // 开启端口
  app.listen(API_PORT, () => console.log(`LISTENING ON PORT ${API_PORT}`));
```

这是后台基础代码，为便于理解，代码中的注释解释了如何链接数据库、读取数据库内容等。关于创建数据库的部分将在下一小节中介绍。

13.1.3 创建数据库

我们使用 Mlab 提供的免费 MongoDB 服务。首先在浏览器中打开 https://mlab.com/ 注册

账号，Mlab 提供了 500MB 免费数据服务。注册账号并登录，单击新建按钮（Create new），如图 13-2 所示。

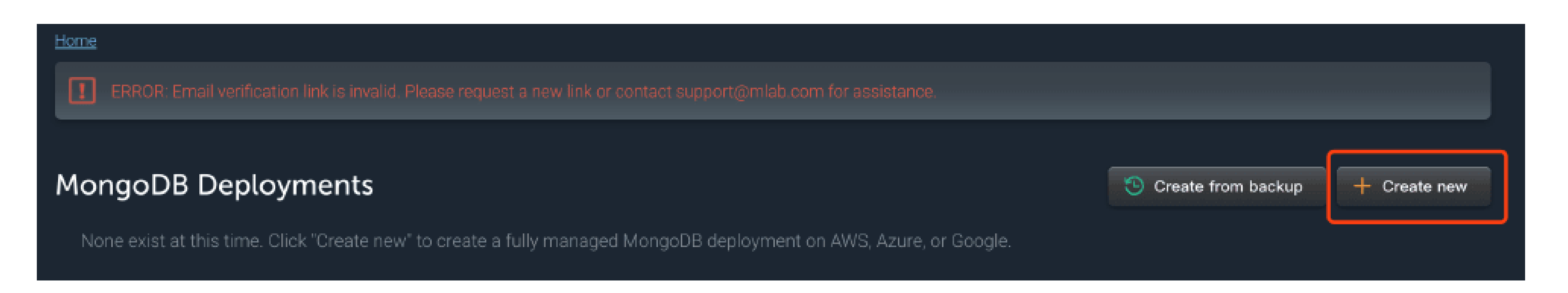


图 13-2　创建数据库

然后选择 amazon web services，如图 13-3 所示。

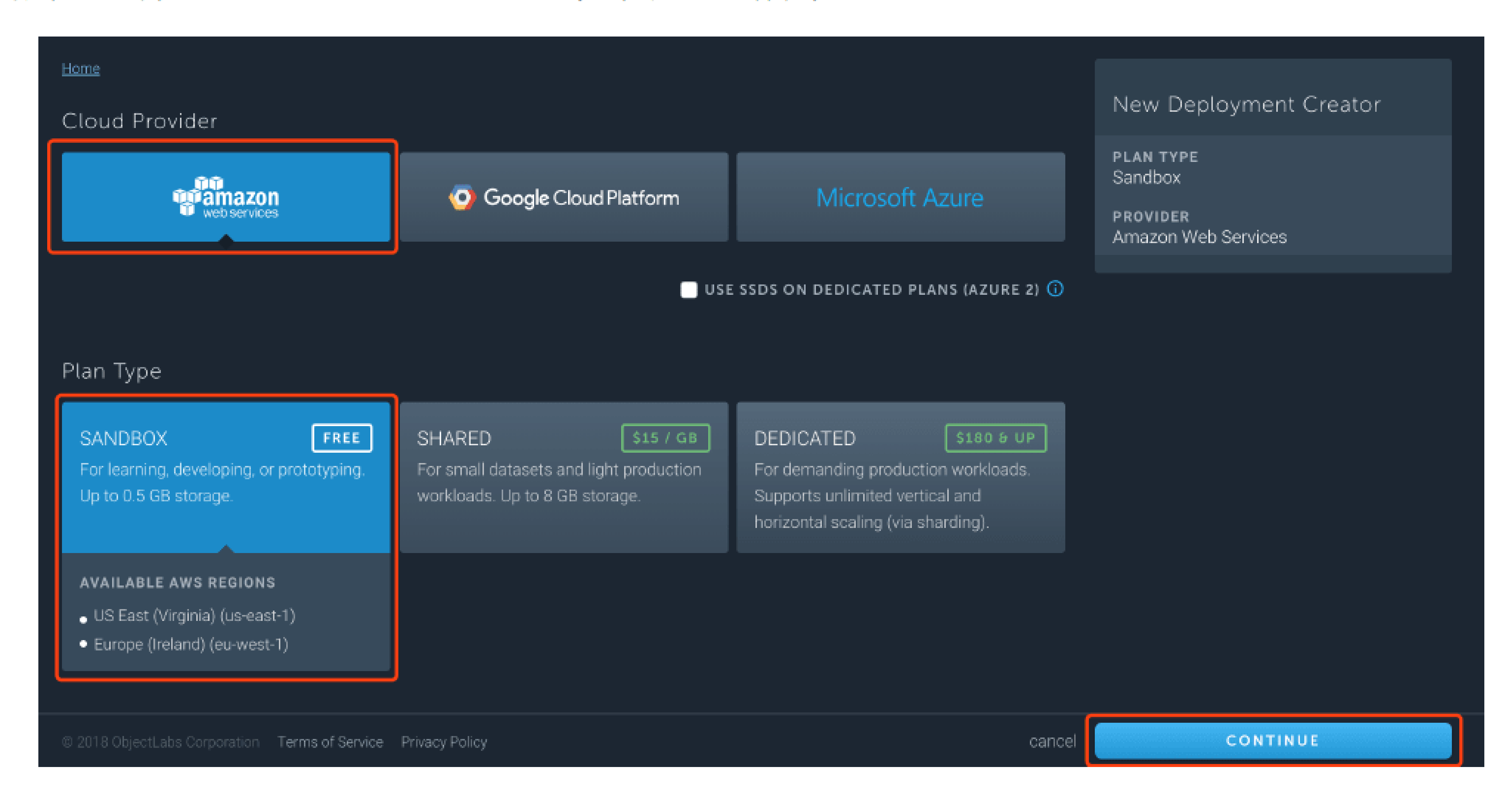


图 13-3　选择云服务提供商

任意选择一个 AWS 区域，单击继续（CONTINUE）按钮，如图 13-4 所示。

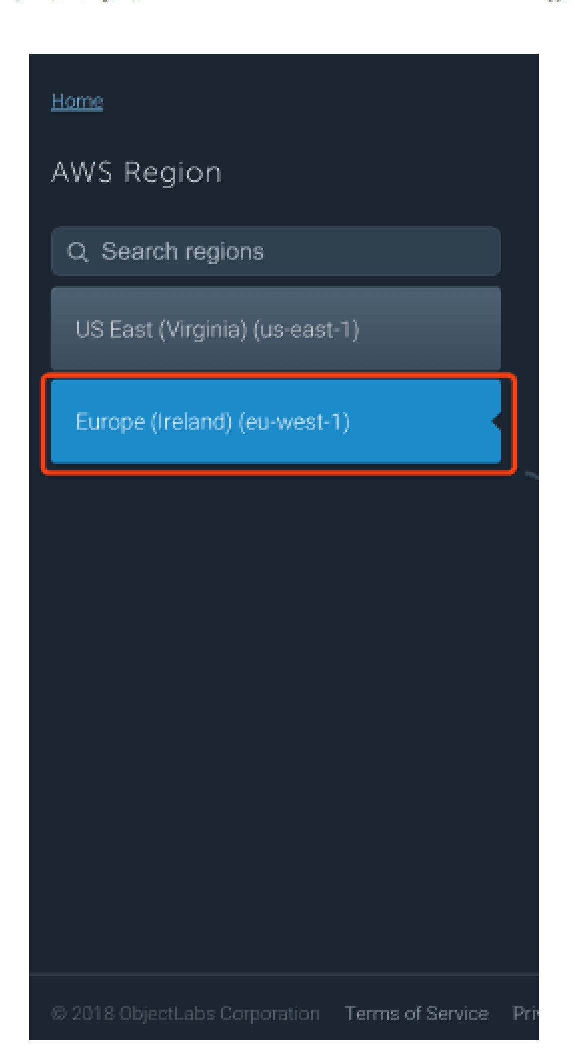


图 13-4　选择 AWS

输入数据库名称，单击继续（CONTINUE）按钮，如图 13-5 所示。

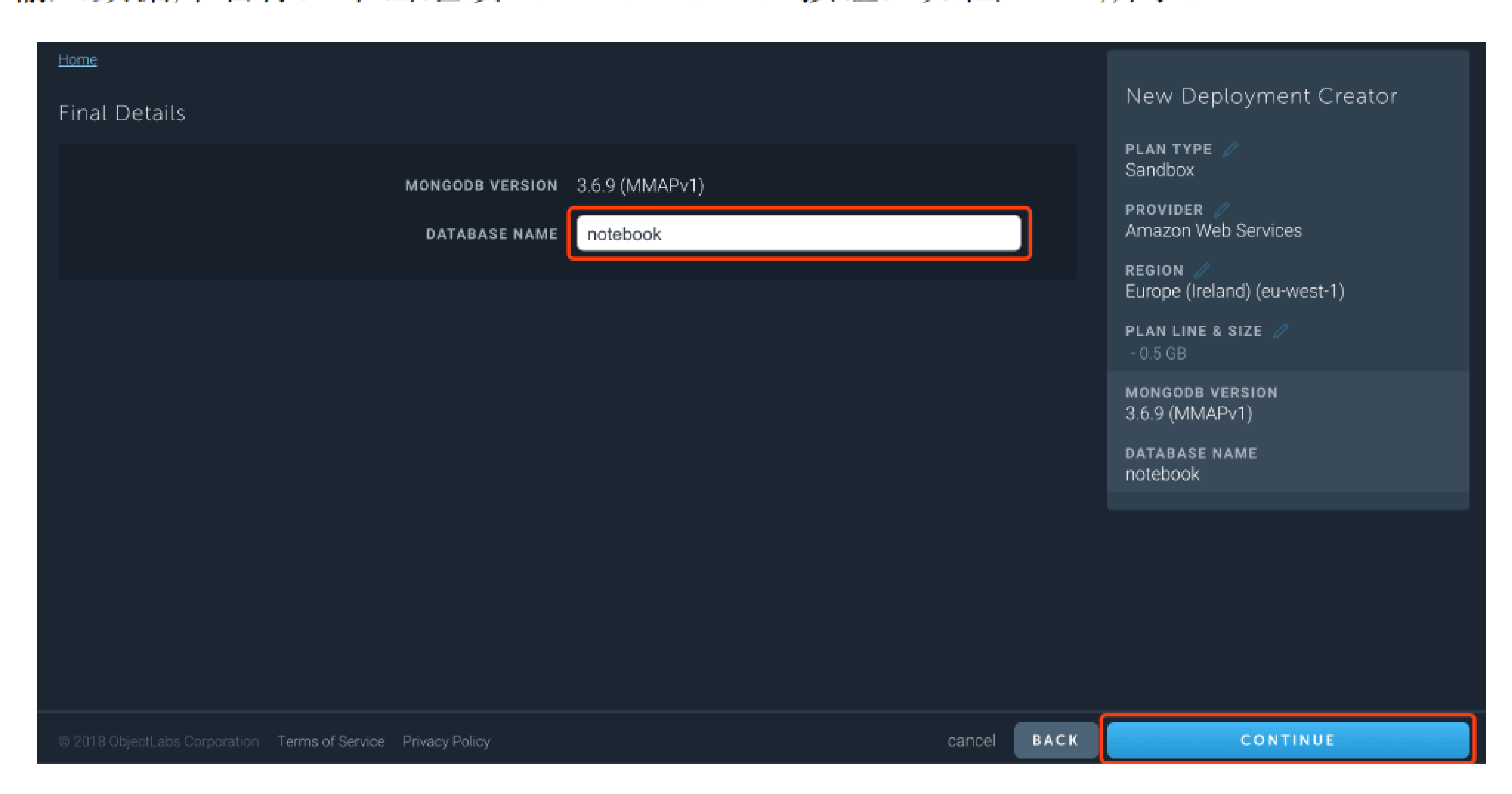


图 13-5　填写数据库名称

提交之后，回到 Mlab 主页，复制 database 信息用于链接数据库。单击 Users 标签，创建用户（Add database user），如图 13-6 所示。

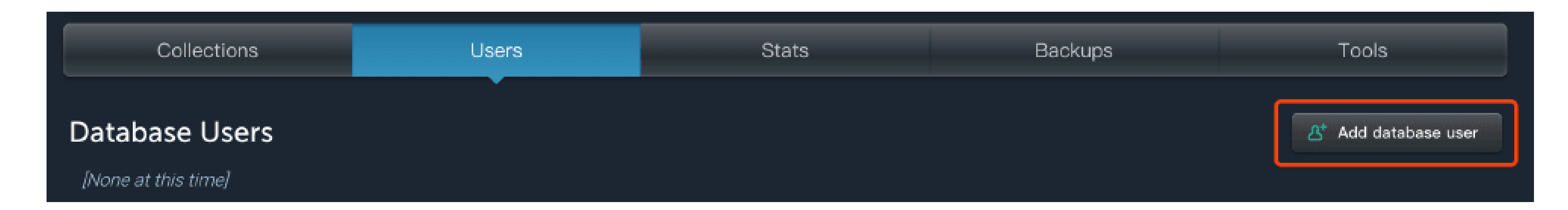


图 13-6　创建用户

回到后台程序中，创建 data.js 文件，输入如下代码：

```
// /backend/data.js
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

// 注意这是我们的数据结构
const DataSchema = new Schema(
  {
    id: Number,
    message: String
  },
  { timestamps: true }
);

//返回 Schema，便于通过 Node.js 使用
module.exports = mongoose.model("Data", DataSchema);
```

安装相关的依赖：

```
npm i -S mongoose express body-parser morgan
```

启动应用：

```
node server.js
```

在控制台可以看到程序已启动，如图 13-7 所示。

```
➜ notebook_app cd backend
➜ backend node server.js
LISTENING ON PORT 3001
connected to the database
```

图 13-7　启动后台程序

13.1.4　连接数据库

回到前台应用中，修改 App.js 文件如下：

```
// /client/App.js
// 引入 react
import React, { Component } from "react";
// 引入 axios 用于发送异步请求获取数据
import axios from "axios";
// 引入 antd 中相关的组件
import {
  Button,
  Input,
  List,
  Avatar,
  Card,
} from 'antd';

// 引入样式文件
import './App.css';

class App extends Component {
  // 初始化 state
  state = {
    data: [],
```

```
    id: 0,
    message: null,
    intervalIsSet: false,
    idToDelete: null,
    idToUpdate: null,
    objectToUpdate: null
  };

  // 首先从数据库中获取已有数据
  // 然后添加轮询机制，用于检测数据库的数据，当数据发生更新时，重新渲染 UI
  componentDidMount() {
    this.getDataFromDb();
    if (!this.state.intervalIsSet) {
      let interval = setInterval(this.getDataFromDb, 1000);
      this.setState({ intervalIsSet: interval });
    }
  }

  // 在 componentWillUnmount 时销毁定时器
  // 需要及时销毁不需要使用的进程
  componentWillUnmount() {
    if (this.state.intervalIsSet) {
      clearInterval(this.state.intervalIsSet);
      this.setState({ intervalIsSet: null });
    }
  }

  // 在前台使用 ID 作为数据的 key 来辨识所需更新或删除的数据
  // 在后台使用 ID 作为 MongoDB 中的数据实例的修改依据
  // getDataFromDb 函数用于从数据库中获取数据
  getDataFromDb = () => {
    fetch("/api/getData")
      .then(data => data.json())
      .then(res => this.setState({ data: res.data }));
  };

  // putDataToDB 函数用于调用后台 API 接口向数据库新增数据
  putDataToDB = message => {
    let currentIds = this.state.data.map(data => data.id);
    let idToBeAdded = 0;
    while (currentIds.includes(idToBeAdded)) {
      ++idToBeAdded;
    }
```

```
  axios.post("/api/putData", {
    id: idToBeAdded,
    message: message
  });
};


// deleteFromDB 函数用于调研后台 API 删除数据库中已经存在的数据
deleteFromDB = idTodelete => {
  let objIdToDelete = null;
  this.state.data.forEach(dat => {
    if (dat.id == idTodelete) {
      objIdToDelete = dat._id;
    }
  });

  axios.delete("/api/deleteData", {
    data: {
      id: objIdToDelete
    }
  });
};

// updateDB 函数用于调用后台 API 更新数据库中已经存在的数据
updateDB = (idToUpdate, updateToApply) => {
  let objIdToUpdate = null;
  this.state.data.forEach(dat => {
    if (dat.id == idToUpdate) {
      objIdToUpdate = dat._id;
    }
  });

  axios.post("/api/updateData", {
    id: objIdToUpdate,
    update: { message: updateToApply }
  });
};

// 渲染 UI 的核心方法
// 该渲染函数渲染的内容与前台界面展示一致
render() {
  const { data = [] } = this.state;
```

```
        console.log('data', data)
        return (
          <div style={{ width: 990, margin: 20 }}>
            <List
              itemLayout="horizontal"
              dataSource={data}
              renderItem={item => (
                <List.Item>
                  <List.Item.Meta
                    avatar={<Avatar src="https://gw.alicdn.com/tfs/
TB1Hup.wa6qK1RjSZFmXXX0PFXa-1024-1024.jpg" />}
                    title={<span>{`创建时间: ${item.createdAt}`}</span>}
                    description={`${item.id}: ${item.message}`}
                  />
                </List.Item>
              )}
            />
            <Card
              title="新增笔记"
              style={{ padding: 10, margin: 10 }}>
              <Input
                onChange={e => this.setState({ message: e.target.value })}
                placeholder="请输入笔记内容"
                style={{ width: 200 }} />
              <Button
                type="primary"
                style={{ margin: 20 }}
                onClick={() => this.putDataToDB(this.state.message)}
              >添加</Button>
            </Card>
            <Card
              title="删除笔记"
              style={{ padding: 10, margin: 10 }}>
              <Input
                style={{ width: "200px" }}
                onChange={e => this.setState({ idToDelete: e.target.value })}
                placeholder="填写所需删除的 ID"
              />
              <Button
                type="primary"
                style={{ margin: 20 }}
                onClick={() => this.deleteFromDB(this.state.idToDelete)}
              >删除</Button>
```

```
          </Card>
          <Card
            title="更新笔记"
            style={{ padding: 10, margin: 10 }}>
            <Input
              style={{ width: 200, marginRight: 10 }}
              placeholder="所需更新的 ID"
              onChange={e => this.setState({ idToUpdate: e.target.value })}
              />
            <Input
              style={{ width: 200 }}
              onChange={e => this.setState({ updateToApply: e.target.value })}
              placeholder="请输入所需更新的内容"
            />
            <Button
              type="primary"
              style={{ margin: 20 }}
              onClick={() =>
                this.updateDB(this.state.idToUpdate, this.state.updateToApply)
              }
            >更新</Button>
          </Card>
        </div>
      );
    }
  }

  export default App;
```

最后，在前台应用 client 中的 package.json 文件中添加代理指向后台程序运行的端口，实现了自动将"http://localhost:3000" 请求转发到"http://localhost:3001" 的服务器：

```
{
  "name": "client",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "axios": "^0.18.0",
    "react": "^16.5.0",
    "react-dom": "^16.5.0",
    "react-scripts": "1.1.5"
  },
```

```
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
  },
  "proxy": "http://localhost:3001"
}
```

回到 notebook_app 目录，执行如下命令：

```
npm init -y
npm i -S concurrently
```

修改 package.json 文件如下：

```
{
  "name": "notebook_app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "concurrently \"cd backend && node server.js\" \"cd client && npm start\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "concurrently": "^4.0.1"
  }
}
```

在 notebook_app 目录下运行：

```
npm start
```

至此，我们完成了笔记本应用程序的环境创建过程，可以直接运行前台和后台的程序。完成的应用程序效果图如图 13-8 和图 13-9 所示。我们可以在前台查看数据库中存储的内容、添加数据、删除已有的数据。

图 13-8　无数据示例状态

图 13-9　添加数据后的示例效果

13.2 引入 antd

在第 13.1 节中，我们完成了粗略的笔记本应用环境搭建，所运行的效果不涉及任何样式。本节我们将使用 antd 对前台界面进行优化。

首先，安装 antd 依赖：

```
yarn add antd
```

修改 src/App.js，引入 antd 的按钮组件：

```
import React, { Component } from 'react';
// 引入 antd 按钮组件
import Button from 'antd/lib/button';
import './App.css';

class App extends Component {
  render() {
    return (
```

```
      <div className="App">
        <Button type="primary">Button</Button>
      </div>
    );
  }
}

export default App;
```

修改 src/App.css，在文件顶部引入 antd/dist/antd.css：

```
@import '~antd/dist/antd.css';

.App {
  text-align: center;
}

...
```

重新运行程序就能看到页面上已经有了 antd 的蓝色按钮组件（见图 13-10），接下来就可以继续选用其他组件开发应用了。



图 13-10　antd 按钮示例

我们现在已经引入 antd 组件并且成功运行起来了，但是在实际开发过程中还有很多问题，上述示例代码实际上加载了全部的 antd 组件的样式，加载了非常多不必要引入的内容，控制台有警告提示，如图 13-11 所示。

```
⚠ ▶You are using a whole package of antd, please use https://www.npmjs.com/package/babel-plugin-import
 to reduce app bundle size.
```

图 13-11　antd 按需加载警告

因此，我们需要对 create-react-app 的默认配置进行自定义，这里我们使用 react-app-rewired 进行自定义配置。引入 react-app-rewired：

```
yarn add react-app-rewired
```

修改 package.json 中的启动配置：

```
/* package.json */
"scripts": {
   "start": "react-app-rewired start",
   "build": "react-app-rewired build",
   "test": "react-app-rewired test"
}
```

在项目根目录（client 目录）中创建一个 config-overrides.js 文件，用于修改默认配置。

```
module.exports = function override(config, env) {
  // webpack 配置
  return config;
};
```

使用 babel-plugin-import（babel-plugin-import 是一个用于按需加载组件代码和样式的 Babel 插件）。现在我们尝试安装 babel-plugin-import：

```
$ yarn add babel-plugin-import
```

修改 config-overrides.js 文件：

```
const { injectBabelPlugin } = require('react-app-rewired');

module.exports = function override(config, env) {
  config = injectBabelPlugin(
    ['import', { libraryName: 'antd', libraryDirectory: 'es', style: 'css' }],
    config,
  );
  return config;
};
```

然后删除 src/App.css 中全量添加的 @import '~antd/dist/antd.css'; 样式代码，修改为如下格式的引入模块方式：

```
// src/App.js

import React, { Component } from 'react';
import { Button } from 'antd';
import './App.css';
```

```
class App extends Component {
  render() {
    return (
      <div className="App">
        <Button type="primary">Button</Button>
      </div>
    );
  }
}

export default App;
```

最后重启 npm start 访问页面，antd 组件的 js 和 css 代码都会按需加载，在控制台上也不会看到警告信息。

使用 antd 可以自定义主题，需要用到 less 变量覆盖功能。我们可以引入 react-app-rewire 的 less 插件 react-app-rewire-less 来帮助加载 less 样式：

```
yarn add react-app-rewire-less
```

修改 config-overrides.js 文件：

```
const { injectBabelPlugin } = require('react-app-rewired');
const rewireLess = require('react-app-rewire-less');

module.exports = function override(config, env) {
  config = injectBabelPlugin(
   ['import', { libraryName: 'antd', libraryDirectory: 'es', style: true }],
// change importing css to less
    config,
  );
  config = rewireLess.withLoaderOptions({
    modifyVars: { "@primary-color": "#1DA57A" },
    javascriptEnabled: true,
  })(config, env);
  return config;
};
```

这里使用 less-loader 的 modifyVars 来进行主题配置。修改后重启 yarn start，如果看到一个绿色的按钮，就说明配置成功了，如图 13-12 所示。



图 13-12　antd 自定义样式

13.3 改写笔记本样式

根据第 13.2 节中的示例，可以对现有笔记本的样式做调整，使之符合大众审美标准。修改后的示例代码如下：

```
// /client/App.js

import React, { Component } from "react";
// 异步请求方法
import axios from "axios";
// 引入 antd 相关组件
import {
  Button,
  Input,
  List,
  Avatar,
  Card,
} from 'antd';
import './App.css';

class App extends Component {
  // 初始化 state
  state = {
    data: [],
    id: 0,
    message: null,
    intervalIsSet: false,
    idToDelete: null,
    idToUpdate: null,
    objectToUpdate: null
  };

  // 首先从数据库中获取已有数据
```

```
// 然后添加轮询机制，用于检测数据库的数据，当数据发生更新时，重新渲染 UI
componentDidMount() {
  this.getDataFromDb();
  if (!this.state.intervalIsSet) {
    let interval = setInterval(this.getDataFromDb, 1000);
    this.setState({ intervalIsSet: interval });
  }
}

// 在 componentWillUnmount 时销毁定时器
// 需要及时销毁不需要使用的进程
componentWillUnmount() {
  if (this.state.intervalIsSet) {
    clearInterval(this.state.intervalIsSet);
    this.setState({ intervalIsSet: null });
  }
}

// 在前台使用 ID 作为数据的 key 来辨识所需更新或删除的数据
// 在后台使用 ID 作为 MongoDB 中的数据实例的修改依据
// getDataFromDb 函数用于从数据库中获取数据
getDataFromDb = () => {
  fetch("/api/getData")
    .then(data => data.json())
    .then(res => this.setState({ data: res.data }));
};

// putDataToDB 函数用于调用后台 API 接口向数据库新增数据
putDataToDB = message => {
  let currentIds = this.state.data.map(data => data.id);
  let idToBeAdded = 0;
  while (currentIds.includes(idToBeAdded)) {
    ++idToBeAdded;
  }

  axios.post("/api/putData", {
    id: idToBeAdded,
    message: message
  });
};


// deleteFromDB 函数用于调用后台 API 删除数据库中已经存在的数据
```

```
  deleteFromDB = idTodelete => {
    let objIdToDelete = null;
    this.state.data.forEach(dat => {
      if (dat.id == idTodelete) {
        objIdToDelete = dat._id;
      }
    });

    axios.delete("/api/deleteData", {
      data: {
        id: objIdToDelete
      }
    });
  };

  // updateDB 函数用于调用后台 API 更新数据库中已经存在的数据
  updateDB = (idToUpdate, updateToApply) => {
let objIdToUpdate = null;
// 遍历数据
    this.state.data.forEach(dat => {
      if (dat.id == idToUpdate) {
        objIdToUpdate = dat._id;
      }
    });
    // 更新数据
    axios.post("/api/updateData", {
      id: objIdToUpdate,
      update: { message: updateToApply }
    });
  };

  // 渲染 UI 的核心方法
  // 该渲染函数渲染的内容与前台界面展示一致
  render() {
    const { data = [] } = this.state;
    console.log('data', data)
    return (
      <div style={{ width: 990, margin: 20 }}>
        <List
          itemLayout="horizontal"
          dataSource={data}
          renderItem={item => (
            <List.Item>
```

```
                <List.Item.Meta
                  avatar={<Avatar src="https://gw.alicdn.com/tfs/
TB1Hup.wa6qK1RjSZFmXXX0PFXa-1024-1024.jpg" />}
                  title={<span>{`创建时间：${item.createdAt}`}</span>}
                  description={`${item.id}: ${item.message}`}
                />
              </List.Item>
            )}
          />
          <Card
            title="新增笔记"
            style={{ padding: 10, margin: 10 }}>
            <Input
              onChange={e => this.setState({ message: e.target.value })}
              placeholder="请输入笔记内容"
              style={{ width: 200 }} />
            <Button
              type="primary"
              style={{ margin: 20 }}
              onClick={() => this.putDataToDB(this.state.message)}
            >添加</Button>
          </Card>
          <Card
            title="删除笔记"
            style={{ padding: 10, margin: 10 }}>
            <Input
              style={{ width: "200px" }}
              onChange={e => this.setState({ idToDelete: e.target.value })}
              placeholder="填写所需删除的 ID"
            />
            <Button
              type="primary"
              style={{ margin: 20 }}
              onClick={() => this.deleteFromDB(this.state.idToDelete)}
            >删除</Button>
          </Card>
          <Card
            title="更新笔记"
            style={{ padding: 10, margin: 10 }}>
            <Input
              style={{ width: 200, marginRight: 10 }}
              placeholder="所需更新的 ID"
              onChange={e => this.setState({ idToUpdate: e.target.value })}
```

```
            />
          <Input
            style={{ width: 200 }}
            onChange={e => this.setState({ updateToApply: e.target.value })}
            placeholder="请输入所需更新的内容"
          />
          <Button
            type="primary"
            style={{ margin: 20 }}
            onClick={() =>
              this.updateDB(this.state.idToUpdate, this.state.updateToApply)
            }
          >更新</Button>
        </Card>
      </div>
    );
  }
}

export default App;
```

本例最终效果如图 13-13 所示。

图 13-13　简易笔记本最终展示效果

13.4 案例小结

本章介绍了如何搭建一个简易的笔记本，仅实现了增、删、改、查的简易功能，后续还有诸多功能需要完善，例如用户管理、笔记分享等扩展功能，有待于读者自行完成。

第 14 章

React实战：React+webpack+ES6实现购物车

本章介绍如何使用 React 实现购物车。此购物车使用 create-react-app 创建项目模板，使用 antd 框架中的字体图标。

14.1 前期准备

14.1.1 环境准备

基于 webpack 和 ES6 结合开发前端 React 应用，我们仍继续使用 create-react-app 创建项目模板，首先安装 create-react-app：

```
npm install -g create-react-app
```

安装完成之后，使用下面的命令创建和初始化项目模板：

```
create-react-app shopping_app
cd shopping_app
```

会发现自动创建了 shopping_app 目录，这时可以使用如下命令运行并开发应用：

```
npm start
```

默认情况下，在开发环境下会启动一个服务器，监听在 3000 端口，启动成功会自动打开浏览器，可以立刻看到这个 App 的默认效果，如图 14-1 所示。

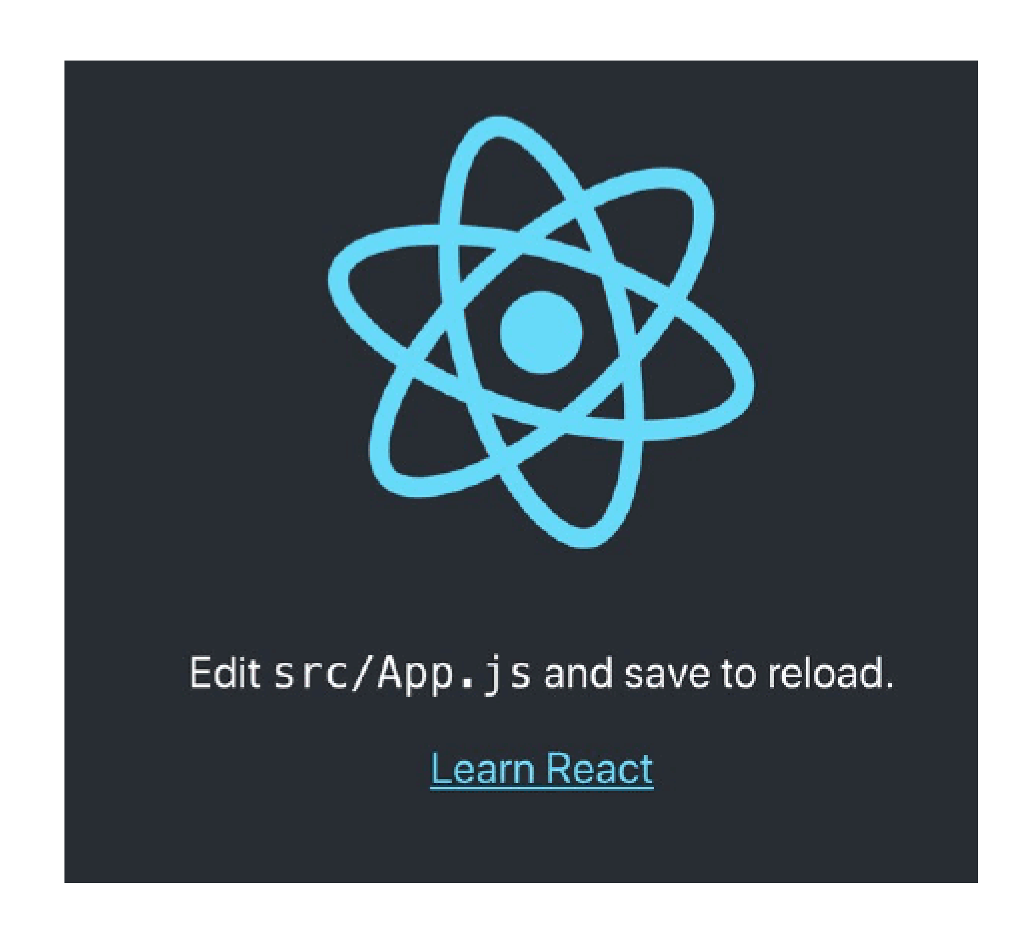


图 14-1　初始化项目预览效果

14.1.2　编码规范 ESLint

形成代码规范的目的是编写高可维护性的前端代码，提升协作和维护效率，不论是对开发思路扩展还是开源项目都有非常大的价值。这里强调代码规范，是因为规范的代码可以促进团队合作、减少缺陷的处理复杂性、降低维护成本、有助于代码审查，同时规范的代码有助于促进开发者自身的成长。

ESLint 是一个语法规则和代码风格的检查工具，可以用来保证写出语法正确、风格统一的代码。ESLint 配置中的规则之间都是独立的，开发者既可以使用默认配置，也可以根据项目的需要进行定制化配置。

首先全局安装 ESLint：

```
npm install -g eslint
```

接着初始化配置文件：

```
eslint -init
```

此时可以选择自己所需的代码风格。ESLint 官方文档做了详细的规则说明，详细的配置请查看 http://eslint.org/docs/user-guide/configuring。

ESLint 代码规范配置示例如下：

```
module.exports = {
    "env": {
        "browser": true,
        "commonjs": true,
        "es6": true
    },
    "extends": "eslint:recommended" ,    //可以选择一些流行的 style 如 airbnb
    "parserOptions": {
        "ecmaFeatures": {
```

```
            "experimentalObjectRestSpread": true,
            "jsx": true
        },
        "ecmaVersion": 7,               // ECMAScript 版本
        "sourceType": "module"
    },
    "plugins": [
        “react”                         //插件，支持 react
    ],
    "rules": {
        "indent": [                     //缩进
            "error”,                    //可选项为 off warn error，对应的数字为 0 1 2
            "space"
        ],
        "linebreak-style": [            //换行 style
            "error",
            "unix"
        ],
        "quotes": [                     // 引号，是单引号还是双引号
            "error",
            "single"
        ],
        "semi": [
            "error",
            "never"
        ]
    }
};
```

ESLint 规范禁止在条件语句（if,while,do...while）中出现赋值操作：

```
// bad
if (user.jobTitle = "manager") {
    // user.jobTitle is now incorrect
}

// good
if (user.jobTitle == "manager") {
    // doSomething()...
}
```

注意，该规则有一个字符串选项，默认是“except-parens”，允许出现赋值操作，但必须是被圆括号括起来的；设置为“always”表示禁止在条件语句中出现赋值语句。

```
// bad 设置为 except-parens
```

```
var x;
if (x = 0) {
    var b = 1;
}

function setHeight(someNode) {
    "use strict";
    do {
        someNode.height = "100px";
    } while (someNode = someNode.parentNode);
}

// good 设置为 except-parens
var x;
if (x === 0) {
    var b = 1;
}
// 设置高度
function setHeight(someNode) {
    "use strict";
    do {
        someNode.height = "100px";
    } while ((someNode = someNode.parentNode));
}
```

禁止在代码中使用 console（在产品发布之前，剔除 console 的调用），注意可以设置 allow 参数允许 console 对象方法：

```
// bad
console.log("Log a debug level message.");
console.warn("Log a warn level message.");
console.error("Log an error level message.")

// good
//自定义的 Console
Console.log("Log a debug level message.");
```

禁止在条件语句（for,if,while,do...while）和三元表达式（?:）中使用常量表达式，可以通过设置 checkLoops 参数来表示是否允许使用常量表达式：

```
// bad
 if (false) {
     doSomething();
```

```
    }
    if (2) {
        doSomething();
    }
    for (;-2;) {
        doSomething();
    }
    while (typeof x) {
        doSomething();
    }
    do{
        doSomething();
    } while (x = -1);
    var result = 0 ? a : b;

   // good
    if (x === 0) {
        doSomething();
    }
    for (;;) {
        doSomething();
    }
    while (typeof x === "undefined") {
        doSomething();
    }
    do{
        doSomething();
    } while (x);

    var result = x !== 0 ? a : b;
```

不允许在函数（function）定义里面出现重复的参数（箭头函数和类方法设置重复参数会报错，但跟该规则无关）：

```
// bad
function foo(a, b, a) {
    console.log("value of the second a:", a);
}
var bar = function (a, b, a) {
    console.log("value of the second a:", a);
};

// good
function foo(a, b, c) {
```

```
    console.log(a, b, c);
}
var bar = function (a, b, c) {
    console.log(a, b, c);
};
```

不允许出现空块语句，但可以通过 allowEmptyCatch 为 true 允许出现空的 catch 子句：

```
// bad
if (foo) {
}

try {
    doSomething();
} catch(ex) {

} finally {

}
//good
if (foo) {
    // empty
}

try {
    doSomething();
} catch (ex) {
    // continue regardless of error
}
```

本章介绍的购物车项目使用的 ESLint 配置示例如下：

```
{
  "extends": "eslint-config-airbnb",
           //可以选择一些流行的 style，如 airbnb standard prettier
  "parser": "babel-eslint",
  "env": {
    "browser": true,
    "es6": true // ECMAScript 版本
  },
  "rules": {
    "comma-dangle": [
      2,
      "only-multiline"
    ],
```

```
    "react/jsx-closing-bracket-location": [
      1,
      "after-props"
    ],
    "no-new": 1,
    "new-cap": [
      "warn",
      {
        "capIsNewExceptions": [
          "CSSModules"
        ]
      }
    ],
    "max-len": 0,
    "no-else-return": 0,
    "eqeqeq": 0,
    "jsx-a11y/img-has-alt": [
      0
    ],
    "array-callback-return": 0,
    "no-plusplus": 0,
    "prefer-arrow-callback": 0,
    "no-bitwise": 0,
    "no-restricted-properties": 0,
    "react/no-unescaped-entities": 0,
    "react/no-unused-state": 0,
    "no-continue": 0,
    "import/prefer-default-export": 0,
    "no-loop-func": 0,
    "no-empty": [
      "error",
      {
        "allowEmptyCatch": true
      }
    ],
    "react/no-find-dom-node": 0,
    "no-cond-assign": 0,
    "react/no-multi-comp": 0,
    "react/prop-types": 0,
    "react/jsx-filename-extension": 0,
    "import/extensions": 0,
    "import/no-unresolved": 0,
    "semi": 0,
```

```
    "linebreak-style": "off"
  }
}
```

开发者可根据需要定制不同的规则。注意 GitHub 开启了提交之前 ESLint 检测规则，因此项目必须配置 ESLint 才可以提交，否则会有 warning，但可以使用 git commit –n 跳过检测，不过不推荐跳过 ESLint 检测。

14.1.3 项目结构

准备好的项目结构如下:

```
|  .babelrc
|  .editorconfig
|  .eslintrc
|  .git
|  .gitignore
|  package.json
|  index.js
|  index.css
|  index.html
|  webpack.config.js
|
├─public
|
├─src
|  ├─src
|  |      Cart.js
|  |      constants.js
|  |      EmptyCart.js
|  |      ProductsContainer.js
|  |      Search.js
|  |      Shopping.js
|  |      utils.js
|  |
|  ├─constants
|  |      ActionTypes.js
|  |
|  └─images
|
└─test
```

由于涉及大量的 ES6 等新属性，因此 Node.js 必须是 10.0.0 以上版本。购物车项目预计所需的功能如图 14-2 所示。

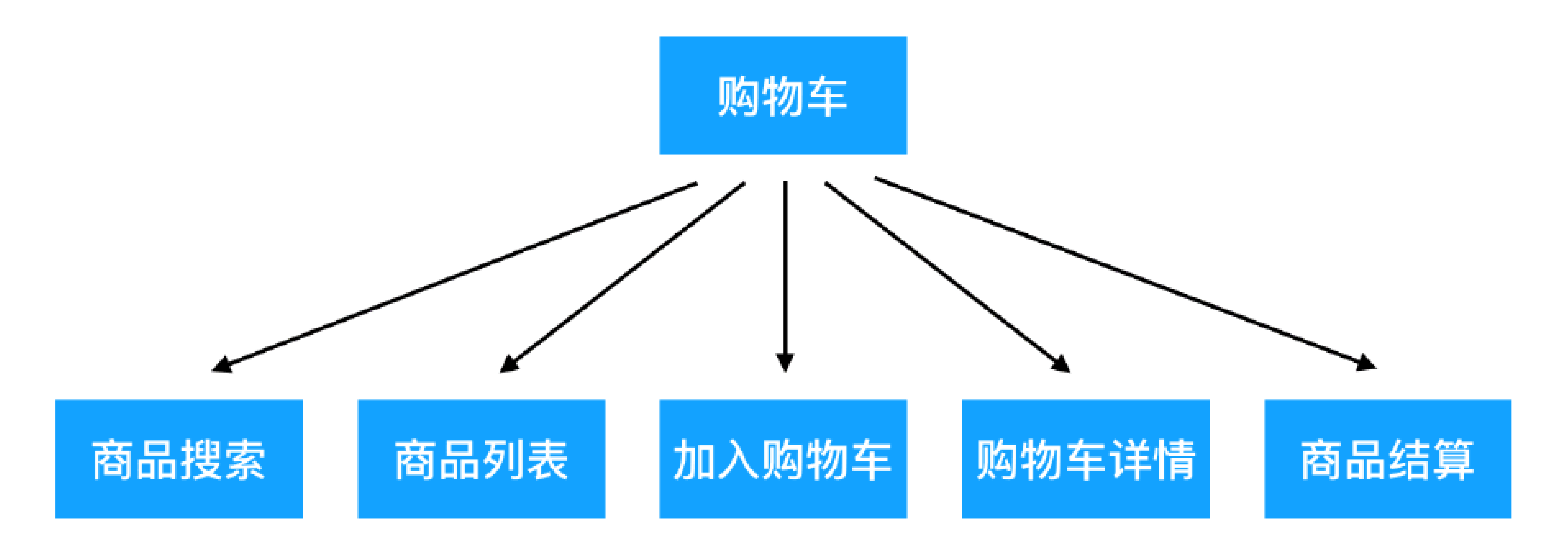


图 14-2　购物车功能设计

14.2 组件设计

14.2.1 购物车框架

首先在 src 文件夹创建 Shopping 文件，Shopping 作为购物车的整体框架：

```
// src/Shopping.js
import React, { Component } from 'react';
// 引入 antd 组件
import {
  Row,
  Col,
} from 'antd';
// 引入购物车组件
import Cart from './Cart';                              // 购物车
// 引入商品列表
import ProductsContainer from './ProductsContainer';    // 商品列表
// 引入商品搜索
import Search from './Search';                          // 商品搜索
// 引入静态数据
import {
  allProducts,                                      // 所有商品
  filterProducts,                                   // 实际展示的商品
  productsModel,                                    // 商品模型
  allQuantity,                                      // 原有库存
  selectQuantity,                                   // 加入购物车的数量
} from './constants'
// 定义购物车应用
class Shopping extends Component {
```

```
  constructor() {
    super();
    this.state = {
      qty: allQuantity,
      products: allProducts,
      selectProducts: [],
      selectQty: selectQuantity,
      filterProducts,
    }
    // 绑定相关事件
    this.searchItem = this.searchItem.bind(this)
    this.handleRemove = this.handleRemove.bind(this)
    this.addCart = this.addCart.bind(this)
  }
  /**
   * 加入购物车
   */
  addCart(index) {
    const currentQty = this.state.qty;
    const selQty = this.state.selectQty;
    const indexNum = index / 1;
// 若库存充足，则购物车数量增加 1
// 若库存不足，则提示用户已售罄
    if (currentQty[indexNum] > 0) {
      currentQty[indexNum]--;
      selQty[indexNum]++;
    } else {
      alert('很抱歉，已售罄！')
    }

    const {
      selectProducts,
      products,
    } = this.state;

    const cart = selectProducts;
    const item = products[indexNum];
cart.push(item.name);
// 更新状态，触发界面更新
    this.setState({
      selectProducts: cart,
      selectQty: selQty,
      qty: currentQty
```

```
      })
    }
    /**
     * 搜索商品
     */
    searchItem(itemName) {
  let finditem = false;
  // 入参为空处理
      if (itemName == '') {
        this.setState({
          filterProducts: productsModel
        });
  } else {
    // 在已有商品中检索
        for (let i = 0; i < productsModel.length; i++) {
          if (productsModel[i].name == itemName) {
            const tmpProducts = [];
            tmpProducts.push(productsModel[i]);
            // 已找到商品，则更新状态
            this.setState({ filterProducts: tmpProducts });
            finditem = true;
            break;
          }
  }
  // 若未找到商品，则提示商品不存在
        if (!finditem) {
          this.setState({
            filterProducts: []
          });
        }
      }
    }
    /**
     * 从购物车中删除
     */
    handleRemove(quantity, id) {
      const originalQty = [10, 8, 15, 5];
      const selProducts = this.state.selectProducts;
      const pname = productsModel[id].name;
      for (let i = 0; i < selProducts.length; i++) {
        const index = selProducts.indexOf(pname);
        if (index > -1) {
          selProducts.splice(index, 1);
```

```
      }
    }
    const selQty = this.state.selectQty;
    selQty[id] = 0;
    const tmpQty = this.state.qty;
tmpQty[id] = originalQty[id];
// 更新购物车状态
    this.setState({
      selectProducts: selProducts,
      selectQty: selQty,
      qty: tmpQty
    })
  }
  /**
   * 渲染 UI
   */
  render() {
    return (
      <div>
        <Row>
          <Col>
            <Search
              searchItem={this.searchItem} />
            <ProductsContainer
              addCart={this.addCart}
              products={this.state.filterProducts}
              qty={this.state.qty} />
          </Col>
        </Row>
        <Row>
          <Cart
            handleRemove={this.handleRemove}
            selectProducts={this.state.selectProducts}
            qty={this.state.selectQty}
            pModel={this.state.products} />
        </Row>
      </div>
    )
  }
}

export default Shopping
```

在总入口文件中引入 Shopping 组件：

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import Shopping from './src/Shopping';
import './index.css';
// 入口 APP 定义
function App() {
  return (
    <div style={{ margin: 100 }}>
      <Shopping />
    </div>
  );
}

ReactDOM.render(<App />, document.getElementById('root'));
```

HTML 文件内容：

```
// index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Demo</title>
  <link rel="stylesheet" href="index.css" />
</head>
<body>
<div id="root"></div>
<script src="common.js"></script>
<script src="index.js"></script>
</body>
</html>
```

14.2.2　商品组件和商品列表

在 src 文件夹下创建 constants.js 文件，输出静态商品数据：

```
// src/constants.js
/**
 * 全部商品数据
 */
export const allProducts = [{
```

```
    index: '0',
    path: 'https://gw.alicdn.com/tfs/TB1UPdcwAPoK1RjSZKbXXX1IXXa-1200-800.jpg',
    name: 'apple',
    price: '4.99',
    quantity: 10
  }, {
    index: '1',
    path: 'https://gw.alicdn.com/tfs/TB1Lr4cwpYqK1RjSZLeXXbXppXa-1200-901.jpg',
    name: 'pear',
    price: '3.59',
    quantity: 8
  }, {
    index: '2',
    path: 'https://gw.alicdn.com/tfs/TB1DUVcwwHqK1RjSZFgXXa7JXXa-788-473.jpg',
    name: 'watermelen',
    price: '6.99',
    quantity: 15
  }, {
    index: '3',
    path: 'https://gw.alicdn.com/tfs/TB15SpewAvoK1RjSZFDXXXY3pXa-1200-800.jpg',
    name: 'banana',
    price: '2.99',
    quantity: 5
  }, {
    index: '4',
    path: 'https://gw.alicdn.com/tfs/TB1hqhswwHqK1RjSZJnXXbNLpXa-1000-669.jpg',
    name: 'kiwi berry',
    price: '8.19',
    quantity: 9
  }, {
    index: '5',
    path: 'https://gw.alicdn.com/tfs/TB13plpwq6qK1RjSZFmXXX0PFXa-1200-797.jpg',
    name: 'pineapple',
    price: '12.19',
    quantity: 20
  }]

  /**
   * 过滤的商品数据
   */
  export const filterProducts = [{
    index: '0',
    path: 'https://gw.alicdn.com/tfs/TB1UPdcwAPoK1RjSZKbXXX1IXXa-1200-800.jpg',
```

```
    name: 'apple',
    price: '4.99'
  }, {
    index: '1',
    path: 'https://gw.alicdn.com/tfs/TB1Lr4cwpYqK1RjSZLeXXbXppXa-1200-901.jpg',
    name: 'pear',
    price: '3.59'
  }, {
    index: '2',
    path: 'https://gw.alicdn.com/tfs/TB1DUVcwwHqK1RjSZFgXXa7JXXa-788-473.jpg',
    name: 'watermelen',
    price: '6.99'
  }, {
    index: '3',
    path: 'https://gw.alicdn.com/tfs/TB15SpewAvoK1RjSZFDXXXY3pXa-1200-800.jpg',
    name: 'banana',
    price: '2.99'
  }, {
    index: '4',
    path: 'https://gw.alicdn.com/tfs/TB1hqhswwHqK1RjSZJnXXbNLpXa-1000-669.jpg',
    name: 'kiwi berry',
    price: '8.19'
  }, {
    index: '5',
    path: 'https://gw.alicdn.com/tfs/TB13plpwq6qK1RjSZFmXXX0PFXa-1200-797.jpg',
    name: 'pineapple',
    price: '12.19',
    quantity: 20
  }]

  /**
   * 商品模型
   */
  export const productsModel = [{
    index: '0',
    path: 'https://gw.alicdn.com/tfs/TB1UPdcwAPoK1RjSZKbXXX1IXXa-1200-800.jpg',
    name: 'apple'
  }, {
    index: '1',
    path: 'https://gw.alicdn.com/tfs/TB1Lr4cwpYqK1RjSZLeXXbXppXa-1200-901.jpg',
    name: 'pear'
  }, {
    index: '2',
```

```
    path: 'https://gw.alicdn.com/tfs/TB1DUVcwwHqK1RjSZFgXXa7JXXa-788-473.jpg',
    name: 'watermelen'
  }, {
    index: '3',
    path: 'https://gw.alicdn.com/tfs/TB15SpewAvoK1RjSZFDXXXY3pXa-1200-800.jpg',
    name: 'banana'
  }, {
    index: '4',
    path: 'https://gw.alicdn.com/tfs/TB1hqhswwHqK1RjSZJnXXbNLpXa-1000-669.jpg',
    name: 'kiwi berry'
  }, {
    index: '5',
    path: 'https://gw.alicdn.com/tfs/TB13plpwq6qK1RjSZFmXXX0PFXa-1200-797.jpg',
    name: 'pineapple',
    price: '12.19',
    quantity: 20
  }]

  /**
   * 商品库存
   */
  export const allQuantity = [10, 8, 15, 5, 9, 20];

  /**
   * 加入购物车的商品数量
   */
  export const selectQuantity = [0, 0, 0, 0, 0, 0];
```

创建商品卡片，在 src 文件夹下创建 Item.js 文件:

```
// src/Item.js

import React, {
  Component
} from 'react'
import {
  Row,
  Button,
  Card,
} from 'antd';

class Picframe extends Component {
```

```
  handleClick() {
    this.props.addToCart(this.props.index);
  }

  render() {
    const {
      name,
      index,
      source,
      quantity,
  } = this.props;

    const popid = name + index;
    return (
      <Card
        bodyStyle={{ padding: 0 }}
        style={{
          width: 302,
          marginTop: 10,
          marginRight: 10
        }}>
        <Row
          type="flex"
          align="middle"
          justify="center"
          style={{ marginBottom: 10, height: 250 }}>
          <img
            src={source}
            style={{ cursor: 'pointer', width: 300 }}
            data-toggle="modal"
            data-target={`#${popid}`} />
        </Row>
        <Row
          style={{ margin: 10 }}>
          <h3>{name}</h3>
          <h4>剩余库存: {quantity}</h4>
        </Row>
        <Row
          align="middle"
          justify="center"
          type="flex"
          style={{ marginBottom: 10 }}>
          <Button
```

```
            onClick={() => this.handleClick()}
            type="primary"> 加入购物车 </Button>
        </Row>
        {/* <div
          className="modal fade"
          id={popid}
          tabIndex="-1"
          role="dialog"
          aria-labelledby="myModalLabel"
          aria-hidden="true">
          <div className="modal-dialog">
            <img src={source} />
          </div>
        </div> */}
      </Card>
    )
  }
}

export default Picframe
```

商品卡片效果如图 14-3 所示。

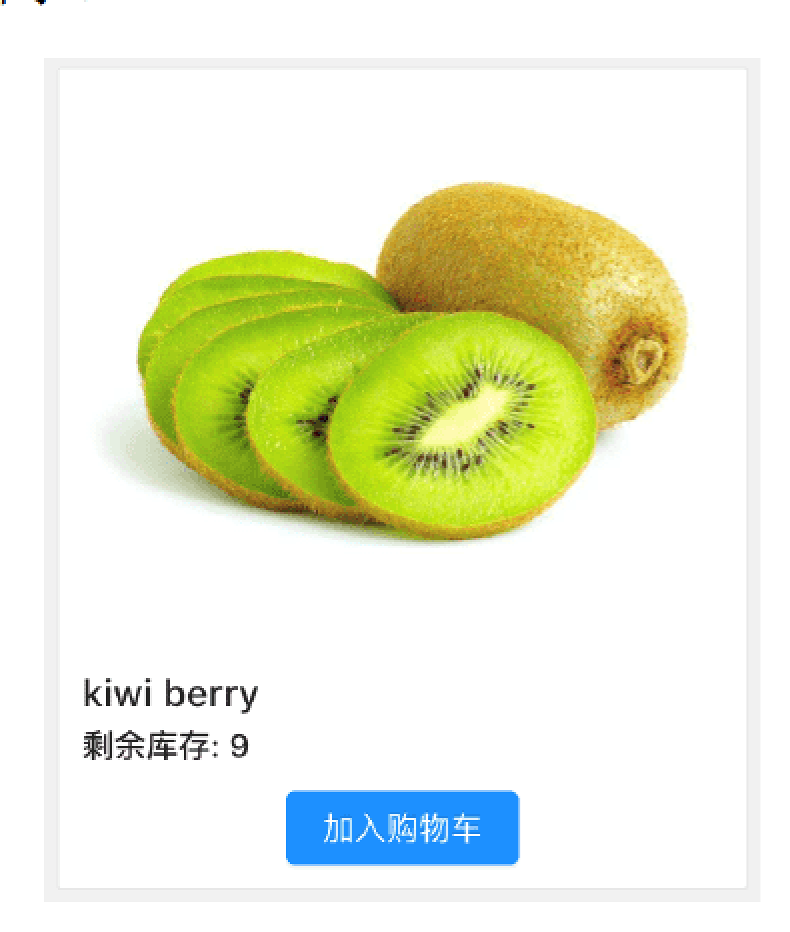


图 14-3　商品卡片

在 src 文件夹下创建 ProductsContainer.js 文件，用于处理商品列表的显示：

```
// 商品列表组件
// src/ProductsContainer.js
// 引入 react
import React from 'react'
// 引入 antd 布局组件
```

```
import {
  Row,
  Card,
} from 'antd';
// 引入商品卡片组件
import Item from './item';
// 引入数组处理方法
import { arrayChunk } from './utils';
// 商品列表
function ProductsContainer(props) {
  // 获取入参
  // 商品列表、库存、加入购物车方法
  const {
    products,
    qty,
    addCart,
  } = props;
  // 对商品列表拆分成一排三
  const goodsList = arrayChunk(products, 3);

  // 渲染商品列表
  if (products.length != 0) {
    return (
      <div>
        {
          goodsList.map((row, rIndex) => (
            <Row
              type="flex"
              key={`row-${rIndex}`}>
              {
                row.map((item, index) => (
                  <Item
                    addToCart={addCart}
                    quantity={qty[item.index]}
                    source={item.path}
                    key={index}
                    index={item.index}
                    name={item.name} />
                ))
              }
            </Row>
          ))
        }
```

```
      </div>
    )
  }
  return (
    <Row>
      <Card style={{ marginTop: 10, width: 920 }}>抱歉，没有找到商品！</Card>
    </Row>
  )
}

export default ProductsContainer
```

为了让商品呈现 3 个一行，使用到数组拆分方法：

```
export default {
  /**
   * @desc 在数组 arr 中取出随机 count 项
   * @param {*} arr 数组
   * @param {*} count 要取出的数据长度
   */
  getRandomArraySlice(arr, count) {
    const newArr = [].concat(arr);
    for (let i = 0, len = newArr.length; i < len; i++) {
      const x = Math.floor(Math.random()) * count;
      // swap
      const tmp = newArr[x];
      newArr[x] = newArr[i];
      newArr[i] = tmp;
    }
    return newArr.slice(0, count);
  },

  /**
   * arrayChunk 方法，把一个数据切分成 size 份数，支持不够的时候自动取随机数进行填充
   * @param {*} arr 数组
   * @param {*} size 要切割的 chunk size
   * @param {*} options 一些拓展参数。比如是否进行自动补全
   */
  arrayChunk(arr = [], size = 4, options) {
    let groups = [];
    if (arr && arr.length > 0) {
      groups = arr.map((e, i) => (i % size === 0 ? arr.slice(i, i + size) :
null)).filter(e => e);
```

```
      }

      if (options && options.autoComplete) {
        const lastIndex = groups.length - 1;
        if (lastIndex >= 0) {
          groups[lastIndex] = groups[lastIndex].concat(
            this.getRandomArraySlice(arr.slice(0, size * lastIndex), size -
groups[lastIndex].length)
          );
        }
      }
      return groups;
    },

  };
```

商品列表效果如图 14-4 所示。

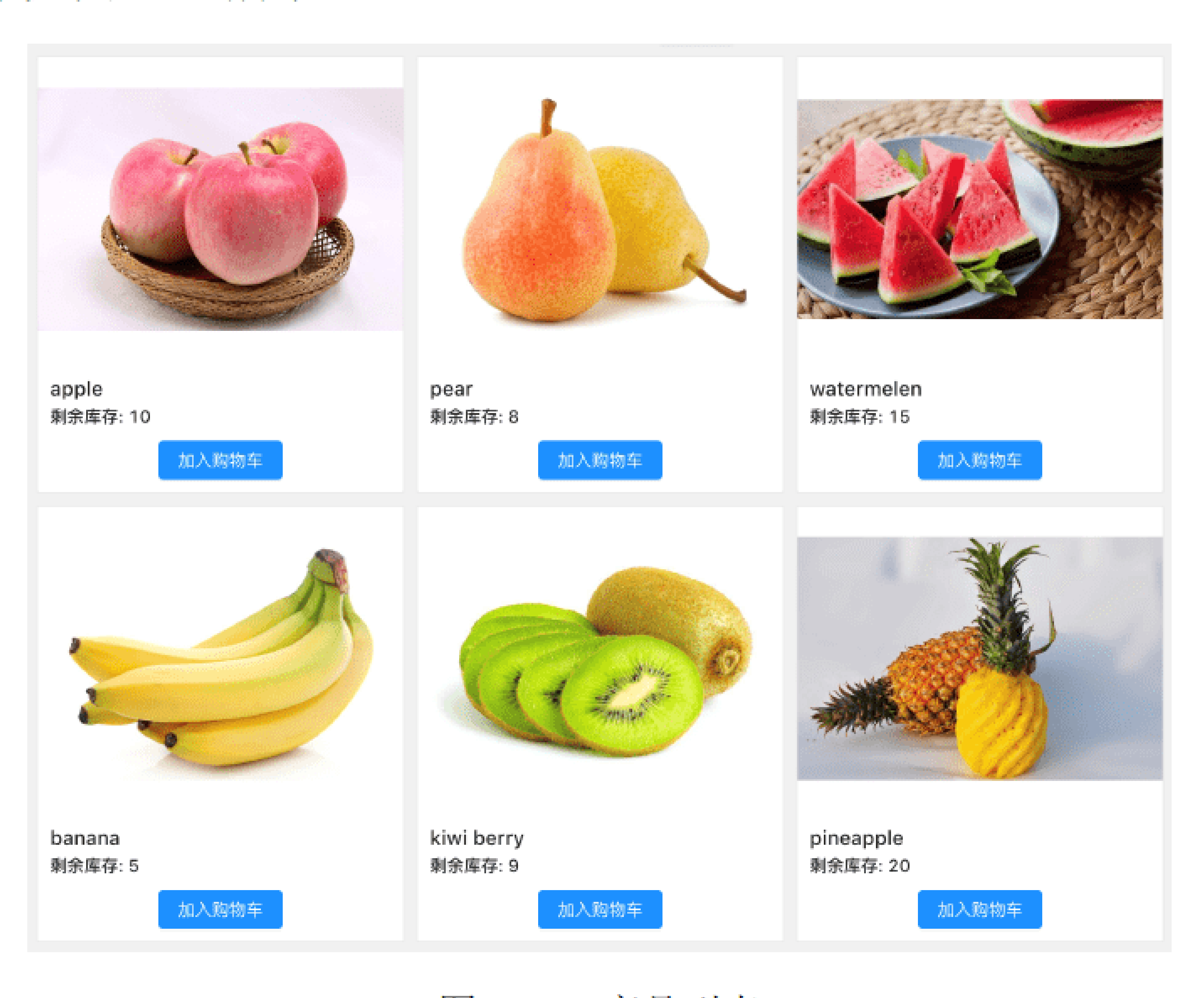


图 14-4　商品列表

14.2.3　商品搜索

在 src 目录下创建 Search.js 文件，实现商品搜索：

```
  // 商品搜索组件
  // src/Search.js

  import React, {
    Component
  } from 'react';
```

```
import {
  Row,
  Col,
  Input,
  Button,
} from 'antd';

class Search extends Component {
  // 搜索按钮单击事件
  handleClick() {
    const search = document.getElementById('search').value;
    this.props.searchItem(search);
  }

  render() {
    return (
      <Row>
        <Col span={12}>
          <Input id="search" placeholder="请输入" />
        </Col>
        <Col span={12}>
          <Button
            type="primary"
            onClick={() => this.handleClick()}
            style={{ marginLeft: 10 }}>搜索</Button>
        </Col>
      </Row>
    )
  }
}

export default Search
```

商品搜索效果如图 14-5 所示。

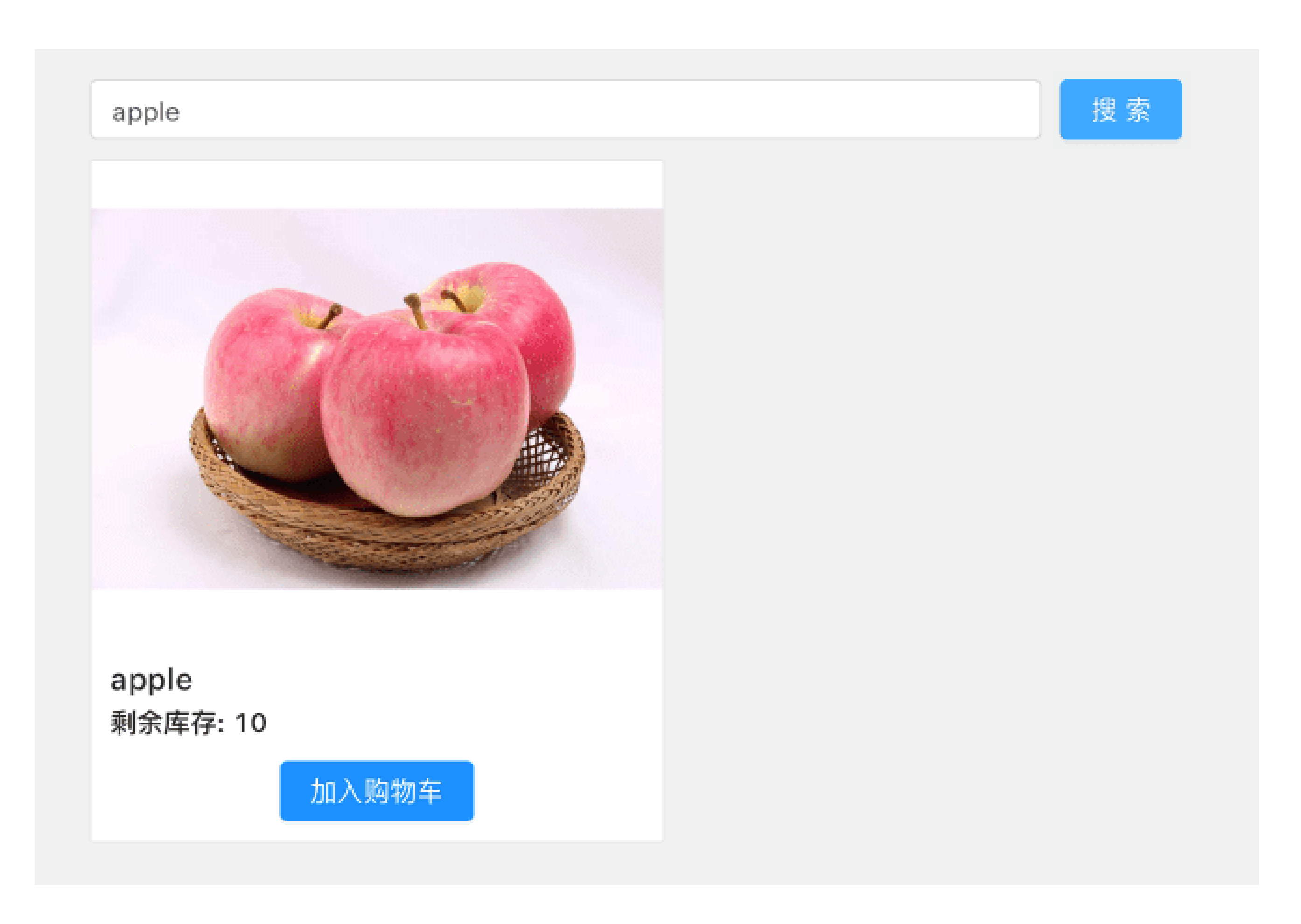


图 14-5　商品搜索组件

14.2.4　购物车

在 src 文件夹下创建 EmptyCart.js 文件，用于处理当购物车中无商品时的提示：

```
// 商品不存在异常提示组件
// src/EmptyCart.js

import React from 'react';
import {
  Card,
} from 'antd'

function EmptyCart() {
  return (
    <Card style={{ marginTop: 10, width: 440 }}>
      <img
        style={{ width: 400 }}
        src="https://gw.alicdn.com/tfs/TB1UmxrwwHqK1RjSZFEXXcGMXXa-658-444.gif"
alt="empty-cart" />
      <h3>您的购物车还是空的，快去添加商品吧！</h3>
    </Card>
  )
}

export default EmptyCart;
```

空的购物车效果如图 14-6 所示。



图 14-6　购物车为空

在 src 文件夹下创建 Cart.js 文件，用于显示/隐藏购物车以及购物车中商品价格的计算：

```
// 购物车组件
// src/Cart.js

import React, {
  Component
} from 'react'
// 引入 antd 布局和按钮组件
import {
  Row,
  Card,
  Button,
} from 'antd'
import EmptyCart from './EmptyCart'

class Cart extends Component {
  constructor(props) {
    super(props);
    this.state = {
      showCart: false,
      cart: [],
      viewChanged: false
    }
    this.handleClick = this.handleClick.bind(this)
    this.handleRemove = this.handleRemove.bind(this)
  }
  // 单击事件
```

```
handleClick() {
  setTimeout(() => {
    this.setState({
      showCart: !this.state.showCart
    })
  }, 0)
}
// 从购物车删除
handleRemove(product, index) {
  this.props.handleRemove(product, index);
  setTimeout(() => {
    this.setState({
      showCart: !this.state.showCart
    })
  }, 100);
  setTimeout(() => {
    this.setState({
      showCart: !this.state.showCart
    })
  }, 200);
}
// 渲染 UI
render() {
  const {
    showCart
  } = this.state;
  const {
    selectProducts,
    qty, // 数量
    pModel,
  } = this.props;
  let cartItems;
  const len = selectProducts.length;
  let totalPrice = 0;

  for (let i = 0; i < qty.length; i++) {
    totalPrice += pModel[i].price * qty[i];
  }
  if (len !== 0) {
    cartItems = qty.map((product, index) => {
      if (product === 0) {
        return null;
      }
```

```
        return (
          <Card key={pModel[index].name}>
            <Row type="flex">
              <img
                style={{ cursor: 'pointer', width: 300, height: 250 }}
                src={pModel[index].path} />
              <div style={{ marginLeft: 10 }}>
                <p>标题: {pModel[index].name}</p>
                <p>价格: {pModel[index].price}</p>
                <p>数量: :{product}</p>
                <p>共计:{product * pModel[index].price}</p>
                <Button
                  style={{ marginTop: 10 }}
                  onClick={() => this.handleRemove(product, index)}>删除</Button>
              </div>
            </Row>
          </Card>
        )
      });
    }

    return (
      <div style={{ padding: '100px 50px 10px' }}>
        <Button
          id="popbtn"
          type="Primary"
          className="btn btn-success"
          onClick={() => this.handleClick()}>我的购物车</Button>
        <Row>
          {
            showCart && len === 0 ? <EmptyCart /> : null
          }
          {
            showCart && len !== 0 ? <div>
              <h3>共计: {totalPrice} 元</h3>
              <div>{cartItems}</div>
            </div> : null
          }
        </Row>
      </div>
    )
  }
}
```

```
export default Cart
```

我的购物车效果如图 14-7 所示。

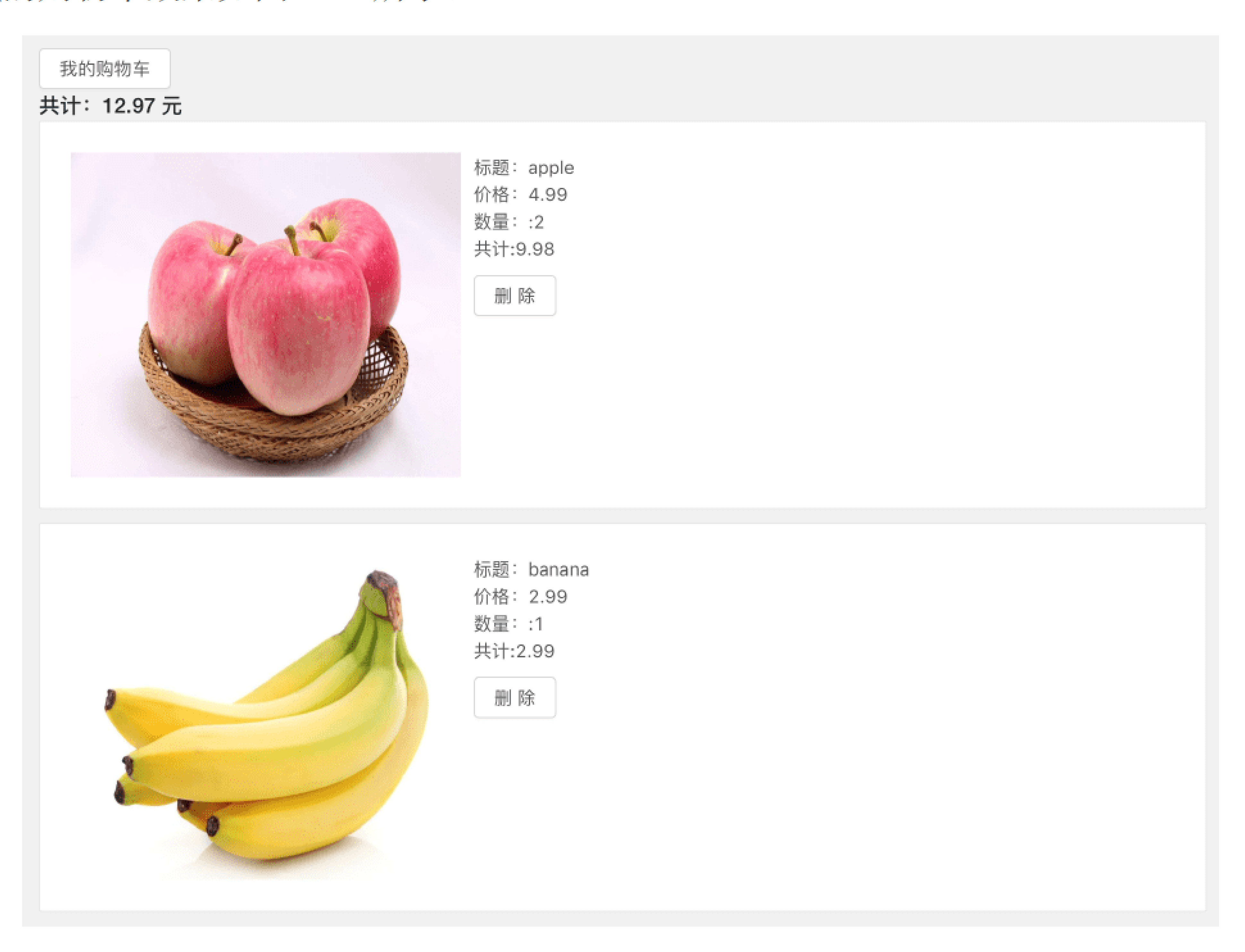


图 14-7　购物车商品及价格统计

14.3 案例小结

在本章介绍的案例中，简单的状态通过组件层层传递，由 Shopping 到 ProductsContainer，再到 Cart，最后到 Item。读者可以自行尝试引入 Redux 进行状态管理，此处不做赘述。